Online Learning on the Programmable Dataplane

Kyle Andrew Simpson

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

School of Computing Science

College of Science and Engineering
University of Glasgow

October 2022

Abstract

This thesis makes the case for managing computer networks with *data-driven methods*—automated statistical inference and control based on measurement data and runtime observations—and argues for their tight integration with *programmable dataplane* hardware to make management decisions faster and from more precise data. Optimisation, defence, and measurement of networked infrastructure are each challenging tasks in their own right, which are currently dominated by the use of hand-crafted heuristic methods. These become harder to reason about and deploy as networks scale in rates and number of forwarding elements, but their design requires expert knowledge and care around unexpected protocol interactions. This makes tailored, per-deployment or -workload solutions infeasible to develop. Recent advances in machine learning offer capable function approximation and closed-loop control which suit many of these tasks. New, programmable dataplane hardware enables more agility in the network—runtime reprogrammability, precise traffic measurement, and low latency on-path processing. The synthesis of these two developments allows complex decisions to be made on previously unusable state, and made quicker by offloading inference to the network.

To justify this argument, I advance the state of the art in data-driven defence of networks, novel dataplane-friendly online reinforcement learning algorithms, and in-network data reduction to allow classification of switch-scale data. Each requires co-design aware of the network, and of the failure modes of systems and carried traffic. To make online learning possible in the dataplane, I use fixed-point arithmetic and modify classical (non-neural) approaches to take advantage of the SmartNIC compute model and make use of rich device-local state. I show that data-driven solutions still require great care to correctly design, but with the right domain expertise they can improve on pathological cases in DDoS defence, such as protecting legitimate UDP traffic. In-network aggregation to histograms is shown to enable accurate classification from fine temporal effects, and allows hosts to scale such classification to far larger flow counts and traffic volume. Moving reinforcement learning to the dataplane is shown to offer substantial benefits to state-action latency and online learning throughput versus host machines; allowing policies to react faster to fine-grained network events. The dataplane environment is key in making reactive online learning feasible—to port further algorithms and learnt functions, I collate and analyse the strengths of current and future hardware designs, as well as individual algorithms.

I'm astounded whenever I finish something. Astounded and distressed. My perfectionist instinct should inhibit me from finishing; it should inhibit me from even beginning. But I get distracted and start doing something. What I achieve is not the product of an act of will but of my will's surrender. I begin because I don't have the strength to think; I finish because I don't have the courage to quit. This book is my cowardice.

—*Fernando Pessoa,* The Book of Disquiet *(p. 156)*

Acknowledgements

First of all, I'd like to thank my wife and best friend Vanessa, for her tireless support and persistence in hearing my a) writing woes, b) angry tirades about software in general, and c) despair over experiment fixes running long into the night—most often in the run up to conference deadlines. I couldn't have made it this far without you. I also owe my family an inestimable debt for their their lifelong help and guidance: to my parents Elaine and Ronnie for always being there and always looking our for me (including making a mid-pandemic move-in possible), my brother Rhys for being there to bounce ideas off of (and hold many technically-involved emulator discussions), and my sister Brooklyn for her constant support. You've all always believed in me and in what I could achieve, and I can't ever pay that back. Lastly, Charlie[1], the best cat, for providing many loud, quizzical meows and for bestowing upon me the highest honour as Chief Feeder.

[1] Unabbreviated: Beloved Sir Charles of House Cattington, Esq., first of his name.

From Netlab, I'd like to thank Prof. Dimitrios Pezaros for his guidance, trust, and advice over the years, as well as the rest of my supervisory team—Simon Rogers and Angelos Marnerides—for their input. To the other Netlab PhD students in particular, Mircea Iordache-Şică, Stefanos Sagkriotis, and Haruna Umar Adoga: our daily meetings have been an essential wellspring of emotional support for me all through the pandemic. It's been great to look out for one another, in terms of morale and to keep engaged and challenged on what we're each working on. Not to mention the many proof-readings of various lumps of text, lovingly provided free-of-charge!

For many a fine lunch at *The Wee Curry Shop* and vibrant discussion (albeit on algorithmics and academic politics!), I have to thank Patrick Prosser, Ciaran McCreesh, James Trimble, Craig Reilly, Blair Archibald, and Ruth Hoffmann. To Patrick in particular, thank you for showing me that research was somewhere I could be and succeed in, and for helping put me down this path.

I'd like to thank all of the co-residents of F101 through the years; Yousef Alhaizaey, Dhahi Alshammari, Dejice Jacob, Thomas Koehler, Lito Michala, Sangkyu Park, Adrian Ramsingh, and Cris Urlea. The discussions, feedback on ideas and GLASS talks, technical help, and an occasionally borrowed poster tube have all been greatly appreciated.

I've met and known so many amazing people through the department via

lunches, cryptic crosswords, puzzles, and Friday coffees that are but a distant memory now. A lot of us started around the same time—or at least met constantly through boardgames, D&D, and mutual friends: Benjamin Bumpus, Marco Cook, Frances Cooper, Patrizia Di Campli San Vito, Natascha Harth, Ellen & William Kavanagh, Charlie Rutherford, Jess Ryan, Gözel Shakeri, Lovisa Sundin, and Tom Wallis. Studying was that much livelier for having you around, and you made Lilybank Gardens somewhere worth going for 5 years. To Andrew Doctor, Sami Kelly, Shaun Lithgow, Callum Milne, Jen Patrick, and Simon Tait: thank you for being lifelong friends (and keeping me sane and socialised!). There are so many other folks I've met through the department, Waterdeep, tea society, and teaching that have been great all around: Vivian Band, Simon Fowler, Katie Heeps, George Holt, Shaun Macdonald, David Maxwell, Michael McKay, Murray McKinstray, Flora McNulty, Stephen McQuistin, Alex Pancheva, Iulia Paun, Will Pettersson, Rebecca Rae, Jenny Savage, Michel Steuwer, Ritchie Walker, and Mihail Yanev.

To Derek 'Del' Hamilton, Douglas MacFarlane, Stewart MacNeill, Robert Nugent and the SoCS support staff, thanks for putting up with our unceasing demands around keeping G131 setup with whatever shiny toys we happened to acquire. To Helen Border and Gail Reat, thanks for keeping teaching and demonstrating as such a tightly run ship.

I met some fantastic people at ESnet, whom I must thank for an incredible 3 month stint in Berkeley, exciting work, and great hills to run on. Crucially, you helped reshape my research direction and introduced me to the joys of programmable dataplane hardware. Richard Czivá, Chin Guok, Yatish Kumar, Bruce Mah, Inder Monga; I hope to work with you again one day soon.

Between various online pastimes—namely, FFXIV and all manner of tabletop shenanigans—I've known and sometimes drifted away from a great number of folk who've offered distant support in their own way. Thank you Bernhard, Cam, Cameron, Carlos, Charles, Fred, Ian, James, Katie, Kyle, Lily, Mal, Olivia, Sasha, Sphenio, and Ves. For routinely keeping me up 'til 2AM—and later—but making it all worthwhile in exchange.

Additional thanks go to Dimitrios Pezaros, Jeremy Singer, Haruna Umar Adoga, Ian Salvagio, my father Ronnie Simpson, and Mircea Iordache-Şică for proof-reading part or all of this thesis. I would like to also thank Colin Perkins, Gianni Antichi, and Jonathan Grizou for the technical, challenging, and fun viva—and by extension for also reading this (very long!) thesis.

And at last, a begrudging 'thanks' to my anonymous reviewers over the years, who have been ever as much my mentors as they have my torturers and gaolers. For the *constructive* advice, I can only thank you.

Original Publications

This thesis is based on the following publications:

- Simpson, K. A., Rogers, S., & Pezaros, D. P. (2020). Per-Host DDoS Mitigation by Direct-Control Reinforcement Learning. *IEEE Trans. Network and Service Management*, *17*(1), 103–117. https://doi.org/10.1109/TNSM.2019.2960202
- Simpson, K. A., Cziva, R., & Pezaros, D. P. (2020). Seiðr: Dataplane Assisted Flow Classification Using ML. *IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, Taiwan, December 7-11, 2020*, 1–6. https://doi.org/10.1109/GLOBECOM42002.2020.9348063
- Simpson, K. A., & Pezaros, D. P. (2021). Online RL in the programmable dataplane with OPaL. In G. Carle & J. Ott (Eds.), *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021* (pp. 471–472). ACM. https://doi.org/10.1145/3485983.3493345
- Simpson, K. A., & Pezaros, D. P. (2022). Revisiting the Classics: Online RL in the Programmable Dataplane [SICSA PhD Conference Best Paper '22]. *2022 IEEE/IFIP Network Operations and Management Symposium, NOMS 2022, Budapest, Hungary, April 25-29, 2022*, 1–10. https://doi.org/10.1109/NOMS54207.2022.9789930

Code, data, and past writings for chapters 4 and 5 are publicly available in:

- https://github.com/FelixMcFelix/rln-dc-ddos-paper,
- https://github.com/FelixMcFelix/pdp-rl-paper.

Code and data used to analyse protocol distributions in *Center for Applied Internet Data Analysis* (CAIDA) traces (appendix A) are publicly available in https://github.com/FelixMcFelix/caida-stats, and code for the microbenchmark used to generate fig. 2.5 is available in https://github.com/FelixMcFelix/xdp-perf-tester. Opus traffic generation code and traces are available at https://github.com/FelixMcFelix/opus-voip-traffic, and capture code is part of https://github.com/FelixMcFelix/felyne-bot (appendix B).

Notes on Typesetting

Body text is set in 11 pt *Libertinus Serif*, a free, libre, and more visually pleasing Times New Roman alternative widely used in ACM publications. Headings, the table of contents, and some text elements (such as complexity classes) are set in sans-serif using the free, humanist *Alegreya Sans* typeface. Numerals, outside of certain mathematical contexts, are set in old-style figures.

This thesis is typeset, first and foremost, around readability. As such, the main body width is set to 11.75 cm, which provides a comfortable 60–65 characters per line. The remaining page width up to the University specification (15.5 cm) is used to provide lengthier asides in the margins[1], which I find considerably less disruptive than footnotes. Line-spacing is set to 136.8 % to compensate, which achieves similar words per page to the University standard (490–520 without paragraph breaks, excluding sidenotes). *Practical Typography* by Matthew Butterick has been an excellent guide in shaping these design choices.

[1] Much like this one. This allows for more interesting comments and/or explanations without disrupting the main flow of the document.

Some colour coding is used for links and markers of various classes:

- Citations to other referenced works,
- Section and Acronym references, which are all functioning links,
- URLs, particularly in the bibliography,
- and Markers indicating key claims in the thesis statement (e.g., *S1*) or key elements of other works' design which benefit from glance value (e.g., $\mathcal{S}$, $\mathcal{A}$).

Contents

List of Tables

List of Figures

List of Algorithms

Acronyms

ABR Adaptive Bitrate 70, 94, 97

ALU Arithmetic Logic Unit 34, 54, 93, 154, 167, 168, 178

API Application Programming Interface 18, 28, 43, 134, 166, 196, 224

APT Advanced Persistent Threat 118, 119

AQM Active Queue Management 53, 92, 174

AS Autonomous System 14, 16, 17, 19, 22, 31, 36, 37, 59–61, 63, 64, 67, 112–117, 119, 122, 123, 134, 152, 174, 188, 189, 199, 212

ASIC Application-Specific Integrated Circuit 1, 12, 13, 15, 16, 22, 25, 26, 31, 34, 37, 39, 40, 48, 91, 129, 175, 198

ATM Asynchronous Transfer Mode 18

BGP Border Gateway Protocol 19, 21, 63, 115–117

BNN Binarised Neural Network 55, 69, 75, 93, 148, 175, 187, 213

CAIDA Center for Applied Internet Data Analysis v, 9, 115, 124, 215, 216

CAM Content-Addressable Memory 229

CBR Constant Bitrate 68, 114–116, 139, 223

CCA Congestion Control Algorithm 2, 3, 7, 8, 30, 40, 50, 52–54, 59, 61, 64–66, 94, 95, 97, 114, 115, 192, 199–201, 203–205, 208, 213

CDN Content Delivery Network 14, 29, 36, 72

CGRA Coarse-Grained Reconfigurable Array 35, 187

CISC Complex Instruction Set Computer 16

CNN Convolutional Neural Network 62–64, 70, 81, 96, 102, 104, 192, 200–206, 208

CPP Command Push-Pull 184, 227

CPU Central Processing Unit 2, 3, 14–16, 23, 25, 26, 31, 34, 35, 38, 40–42, 46, 47, 54, 55, 65, 66, 71, 72, 76, 77, 92, 105, 112, 147, 171, 174, 177, 178, 180, 196, 202, 214

CSP Constraint Satisfaction Problem 63, 76

DAG Directed Acyclic Graph 71, 72

DASH Dynamic Adaptive Streaming over HTTP 70, 215

DDN Data-Driven Networking 3–7, 21, 49, 54–56, 59–62, 67, 76–78, 82, 89, 90, 94–98, 106, 109, 111, 116, 145, 147–149, 191, 209–212

DDoS Distributed Denial of Service 2, 5–8, 22, 36, 51, 61, 65, 67, 68, 98, 109–112, 114, 116–119, 127, 134, 135, 142, 144, 145, 149, 152, 162, 174, 177, 187, 188, 190, 209, 210, 212

DMA Direct Memory Access 40, 54, 148

DNN Deep Neural Network 4, 55, 63, 65, 70, 72, 74–78, 82–84, 93, 97, 98, 100, 103, 148, 150, 211, 212

DNS Domain Name System 111, 113–116

DoS Denial of Service 98, 102, 111, 112

DOTS DDoS Open Threat Signalling 117, 123, 127

DPDK Data Plane Development Kit 24, 25, 28, 40–42, 44–46, 48, 183, 199

DPI Deep Packet Inspection 38, 52, 64

DRAM Dynamic Random-Access Memory 162

DRL Deep Reinforcement Learning 62, 63, 65, 66, 69–73, 77, 82, 97, 100, 147, 183

DSCP Differentiated Services Code Point 56, 169, 231

DSL Domain-Specific Language 46, 69

eBPF Extended Berkeley Packet Filter 16, 28, 34, 41–45, 47, 48, 214

ECMP Equal-Cost Multi-Path routing 52, 62, 122

ES Evolution Strategies 63, 88, 213

FCT Flow-Completion Time 53, 62, 65, 85, 115

FL Federated Learning 60, 68, 69, 102, 103

FPGA Field-Programmable Gate Array 26, 32, 33, 35, 47, 48, 54–56, 92, 148, 150, 151, 154, 162, 168, 175, 184, 187, 190, 213, 214

FPU Floating-Point Unit 39, 91–93, 126, 148, 149, 154, 190, 193, 211

FSM Finite State Machine 27, 34, 35, 123, 124, 225

FU Functional Unit 4, 25, 26, 38, 41, 73, 91, 92, 151, 169, 210, 211, 213

GNN Graph Neural Network 63, 69, 71–74, 76, 81, 82, 103

GPU Graphics Processing Unit 3, 35, 54, 55, 64, 66, 76, 81, 105, 147, 157

GRU Gated Recurrent Unit 64, 82

HLS HTTP Live Streaming 70

HTTP Hypertext Transfer Protocol 24, 71, 110, 113, 117, 122, 132–134, 137, 140, 141, 215, 217

IAT Inter-Arrival Time 7, 129, 130, 132, 133, 172, 192, 194, 196, 197, 200–204, 206–208

IDS Intrusion Detection System 22, 24, 25, 30, 64, 69

IETF Internet Engineering Task Force 13, 14, 22

IID independent and identically distributed 105, 109

ILP Integer Linear Programming 24, 49, 62, 63, 73

INT In-band Network Telemetry 29, 50, 51, 53, 191

IoT Internet of Things 93

IP Internet Protocol 51, 73, 112, 113, 116, 120, 123, 131, 132, 134, 153, 196, 216, 226, 231

IPC Inter-Process Communication 151, 159, 160, 165–167

IR Intermediate Representation 32, 46, 48

IRQ Interrupt Request 40, 41, 43, 44

ISA Instruction Set Architecture 16, 26, 47

ISP Internet Service Provider 9, 16, 22, 51, 62, 69, 71, 110, 114–118, 189, 214–217

IXP Internet eXchange Point 16, 19, 36, 217

JCT Job Completion Time 72

JIT Just-in-Time 43, 47, 48

***k*NN** *k*-Nearest Neighbours 64, 118, 200, 202–204, 206, 208

LFA Link-Flooding Attack 6, 51, 114–118, 123

LSTM Long Short-Term Memory 35, 64, 65, 71, 76, 82, 96, 187, 192, 196, 202, 210

LUT Look-up Table 48, 162

MAC Medium Access Control 26, 36, 40, 41, 44, 183, 184, 199, 227

MAT Match-Action Table 4, 12, 20, 27, 28, 31, 33–35, 38, 47, 48, 51–53, 55, 64, 80, 94, 148, 151, 154, 175, 211, 213

MDP Markov Decision Process 61, 74, 85, 86, 88, 90, 91, 95, 106, 152

ME Microengine 33, 160, 161, 164, 167–172, 174, 175, 184, 186, 227–229

MIB Management Information Base 15, 16

ML Machine Learning 3–8, 30, 35, 38, 39, 49, 54–56, 59, 60, 63, 64, 67, 68, 70–72, 74, 76–78, 83–85, 90–100, 104–106, 109, 110, 143, 148, 149, 175, 187, 190–192, 196, 200, 206–209, 211, 214

MPLS Multiprotocol Label Switching 19, 69

MPMC Multi-Producer/Multi-Consumer 162, 165, 168

MPTCP Multipath TCP 65, 70

MTU Maximum Transmission Unit 38, 50, 135, 165, 207

NAT Network Address Translation 22, 117, 224

NF Network Function 23, 24, 36, 38, 39, 46, 188

NFP Netronome Flow Processor 8, 9, 36, 47, 52, 55, 151, 154, 160, 161, 165–167, 172, 174, 175, 178, 184, 186, 187, 198, 201, 213, 227–229

NFV Network Functions Virtualisation 23, 119, 144

NIC Network Interface Card 2, 7, 12, 26, 30, 35, 36, 38, 40–44, 46–48, 54, 55, 69, 75, 78, 90, 93, 94, 147–152, 155, 165, 173–175, 177, 178, 180, 181, 183, 186, 187, 190, 210, 214, 227

NN Neural Network 35, 55, 62–68, 70, 72, 75, 78, 80–82, 89, 93, 96, 97, 100–102, 104, 147, 187, 189, 213

NPU Network Processing Unit 2, 12, 15, 16, 25, 26, 31, 48, 52, 55, 75, 92, 151, 213

NTP Network Time Protocol 113, 135, 224, 226

NUMA Non-Uniform Memory Access 40

OS Operating System 15, 16, 20, 24, 26, 29, 36, 40–43, 68, 111, 113, 214

OSPF Open Shortest Path First 18, 19

OVS Open vSwitch 21, 28, 43, 64, 119, 120, 129, 143

PCIe PCI Express 35, 40, 41, 54, 104, 105, 147–149, 152, 169, 172, 174, 183, 210, 227

PDP Programmable Dataplane 2, 4–8, 12, 13, 22, 29–31, 33, 35–39, 46, 48–56, 75, 76, 78, 80, 82, 83, 88, 90–93, 129, 148–151, 154, 155, 157, 165, 171, 175, 189–193, 196, 207, 209–211, 213, 214, 228

PIFO Push-In First-Out 34, 53

PISA Protocol Independent Switch Architecture 27, 28

PS Parameter Server 56

PSA Portable Switch Architecture 7, 28, 32–36, 46, 55, 150, 152, 192, 193, 196, 198, 208

QoE Quality of Experience 53, 63, 69–71, 75, 123, 223

QoS Quality of Service 3, 4, 24, 34, 35, 53, 54, 61, 63, 75, 122, 191, 194, 212

RAM Random Access Memory 40, 44, 46, 72, 77, 83, 112, 136, 162, 171, 202, 203

RIP Routing Information Protocol 18

RISC Reduced Instruction Set Computer 15, 16, 26, 42

RL Reinforcement Learning 3, 4, 6–8, 56, 59–63, 65–70, 72–74, 76–79, 81–86, 89–91, 94, 95, 97, 100, 102, 106, 107, 109, 110, 116, 117, 119, 120, 124, 126, 127, 135, 138, 139, 142, 144, 145, 147–152, 154–157, 162–166, 172–175, 177, 178, 183–190, 209, 210, 212, 213, 231

RMT Reconfigurable Match Tables 2, 27, 28, 31, 33–36, 55, 56, 80, 82, 198, 213, 214

RNN Recurrent Neural Network 82, 192

RPC Remote Procedure Call 35, 39, 114

RTCP RTP Control Protocol 224–226

RTP Real-time Transport Protocol 134, 223–226

RTT Round-Trip Time 36, 39, 44, 45, 53, 62, 65, 66, 70, 71, 73, 166, 183, 185, 189

SCTP Stream Control Transmission Protocol 22, 114

SDN Software-Defined Networking 1, 2, 6, 12, 13, 18–21, 23, 24, 29, 30, 38, 53, 60, 68, 110, 119, 133, 143–145

SGD Stochastic Gradient Descent 83, 84, 104

SIMD Single Instruction Multiple Data 35, 55, 91, 167–169, 181

SKB Socket Buffer 40–43

SLA Service-Level Agreement 94, 199

SMT Satisfiability Modulo Theories 33, 63, 97

SoC System on a Chip 15, 25, 148, 151, 160, 168, 187, 214, 227

SRAM Static Random-Access Memory 67, 162

SSRC Synchronisation Source 224–226

SVM Support Vector Machine 35, 55, 96, 101, 187

TCAM Ternary Content-Addressable Memory 26, 27, 38, 64, 75, 94, 144, 150, 211

TCP Transmission Control Protocol 7, 17, 24, 40, 49, 53, 65, 66, 68, 110, 112–114, 117, 124, 130–134, 136, 137, 139, 140, 142, 188, 192, 196, 197, 199, 201–204, 208, 215–217, 219

TD Temporal-Difference 86, 87, 89, 157, 159

TE Traffic Engineering 1, 12, 17–19, 21, 22, 52–54, 61, 62, 97

TEE Trusted Execution Environment 25, 105

TLS Transport Layer Security 71, 116

TO Traffic Optimisation 19, 61, 62, 97

TPU Tensor Processing Unit 54, 81, 92, 105

TRS Timed Random Sequential 125, 126, 145, 188, 189

TURN Traversal Using Relays around NAT 135, 223–225

UDP User Datagram Protocol 14, 22, 68, 112–114, 116, 117, 122, 124, 129, 130, 132, 133, 135, 137, 142, 169, 183, 198, 216, 217, 220, 221, 223, 224, 226, 231, 232

VLIW Very Long Instruction Word 25, 27, 47

VM Virtual Machine 15–17, 23, 24, 42

VNF Virtual Network Function 6, 8, 12, 23–25, 36–39, 48, 97, 120, 126–129, 145, 149, 173, 174, 178, 183

VoIP Voice over Internet Protocol 6, 9, 110, 115, 116, 123, 134, 142, 223, 224

VPN Virtual Private Network 64

VRAM Video RAM 202

WAN Wide Area Network 21, 22, 191, 199

XDP eXpress Data Path 39, 41, 43–45, 47, 51, 183, 214

Chapter 1

Introduction

Computer networks are complex, yet critical infrastructure. The Internet of today is a prime example. From its birth as *ARPANET* (Heart *et al.*, 1970), interlinking the computer networks of self-governing research institutions, it has scaled to connect together billions of devices over chains of smaller networks owned by totally separate entities. This growth has come at a cost; large-scale networks are beset with layers upon layers of interlocking and overlaid systems divided amongst endpoint hosts and operators of the network fabric. The isolated, resilient design of all these layers is a strength in itself as these layers can, in theory, be replaced. Yet owing to this complexity, keeping the 'Internet machinery' well-oiled and performant is a difficult task.

As these networks have grown larger and faster over the last half-century, they have become more flexible to do just this. Early design choices such as routing algorithms had been bonded to fixed-function hardware. To escape these shackles, the community sought through the early 2000s to separate the high-level forwarding behaviour of network packets (the *control plane*) from the hardware dataplane, giving rise to *Software-Defined Networking* (SDN). This is a research success that has meaningfully impacted the design, adaptability, performance, and fault-tolerance of production networks. For instance, the ability to make routing decisions per-flow has enabled *Traffic Engineering* (TE) that formerly required complex workarounds. What this did not solve was inflexible *dataplane* behaviour; switches still supported a fixed set of actions applied to a fixed set of protocols known ahead-of-time, with limited shared state for measurement purposes. Many packet actions—security functions and the like—were and are implemented as *Application-Specific Integrated Circuits* (ASICs) inside *middleboxes* to process traffic at line rate. Infamously, these add yet more inflexibility by relying on (possibly incorrect) handling of known protocols. Making up for this shortfall in malleability by implementing these tasks in host machines causes significant, orders-of-magnitude reductions in packet throughput and added latency. Between fixed-function dataplanes and commodity hosts, there is

no way to balance performance and arbitrary programmability. While it is one thing to argue around performance and ossification, what of new functionality? To observe incorrect (yet transient) paths followed by packets, or to inspect and aggregate queue state and nanosecond-level timestamps to detect recent issues such as *microbursts* both require packet modification and access to state which past switches simply do not allow.

Many other aspects of networks are run by heuristic methods: each complex, hand-tuned, and operating on limited information. Their responses must be both approximately correct *and* computationally cheap to scale reasonably. To make the point clear, consider *Congestion Control Algorithms* (CCAs). CCAs underpin the majority of Internet traffic's ability to dynamically scale send rates up or down—in the absence of actual network state, they must in effect reverse-engineer optimal actions by following a proxy metric such as delay or packet losses. While this is impressive, one can't help but ask if better information about the network itself could allow more useful decision making. What also happens, then, as networks change? Experience has shown that we simply iterate, from NewReno (Gurtov *et al.*, 2012) to Cubic (Rhee *et al.*, 2018) to BBR (Cardwell *et al.*, 2016)[1] in long-fat networks. But even after years of design and tuning, these are easy to get subtly or fatally wrong; initial actively-deployed BBR versions were notably unfair to other flows (Ware *et al.*, 2019). The takeaway is that it's infeasible to develop and hand-tune strategies per workload, per topology, and per protocol distribution. Should we not be able to automatically infer tailored mechanisms or parameters, robust to changes and evolution, from local performance and global management data—a *data-driven* solution? CCAs are but one case where we should ask ourselves these questions. Consider general optimisation of network protocols and infrastructure, or protection against the abuse of network resources such as *Distributed Denial of Service* (DDoS) attacks, or even the above dataplane measurement—equally strong candidates to consider whether *data-driven* logic and *network cooperation* can lead to meaningful improvement.

[1] Deployed circa 1999, 2006, and 2016 respectively.

While these aims are lofty, the last decade has seen surprising and rapid kinds of change, first of all in the design and introduction of *programmable* switching hardware and *Network Interface Cards* (NICs). *Programmable Dataplane* (PDP) switch hardware was originally designed to evolve past the fixed action sets of SDN at line rate, using a limited compute model rather than aiming for full programmability on par with host *Central Processing Units* (CPUs). Indeed, the turnaround from the original *Reconfigurable Match Tables* (RMT) proposal (Bosshart *et al.*, 2013) to full-scale switches based on Intel's *Tofino 2* (Intel, 2022) and Nokia's *FP5* (Nokia, 2021), aggregating 12.8–14.4 Tbit/s, in a scant few years is remarkable. Diversifying the field further, the legacy of older *Network Processing Units* (NPUs) has led to *SmartNICs*, offering more expressive and capable compute at a smaller scale such as via Intel's *infrastructure processing units* (Intel, 2021b). As it happens, these tools have not only enabled greater control and adaptability of networks

but also powerful schemes to measure them, a new environment to execute program logic, and tighter cooperation with end hosts. What's fascinating is that *these* ideas and use cases are not entirely novel, reflecting an undercurrent present since the *active networking* movement (Tennenhouse & Wetherall, 1996). Instead, both classes of efficient hardware have revealed the value of *offloading* and *in-network compute*—moving all or part of an application's logic to the network fabric to accelerate it further, in spite of its different compute capabilities versus a typical CPU. Moreover, these new classes of hardware are fully reprogrammable at runtime, allowing line-rate services to be easily installed, upgraded, and replaced.

The second substantial change of the last decade is the meteoric rise of *Machine Learning* (ML) and *Reinforcement Learning* (RL) through high-profile, breakaway successes in difficult domains such as classification (K. He *et al.*, 2016) and game playing (Berner *et al.*, 2019; Silver *et al.*, 2017). These approaches learn a function to map input data (like the statistics of a monitored flow) to output labels or actions, repeatedly transforming it according to complex learnt statistical properties, with the aim that a learnt function extends well from seen to unseen data. This, too, is the revival of an older line of research—statistical and connectionist ideas which have been extended and empowered by powerful, specialised compute resources like commodity *Graphics Processing Units* (GPUs). What this offers us is the necessary toolkit for *Data-Driven Networking* (DDN)—the automatic tuning that networks cry out for—allowing the development of better generalised solutions to network problems, or even policies specifically tailored to the needs of a deployment environment. RL methods in particular have a unique affinity for closed-loop control tasks. These policies are iteratively learnt by taking an action—deliberately exploring supposedly suboptimal choices from time to time—before observing the controlled system's state some time later and using a measured *reward* score to improve the policy itself in its own feedback loop. We have, at last, the tools to learn complex decision boundaries and effective control in spite of the very non-trivial (and often surprisingly involved) system dynamics of computer networks. The beauty is that, even as our networks evolve, we should be able to learn adjustments and corrections to account for new protocols, behaviours, and topology changes by learning from (always-available) performance and *Quality of Service* (QoS) metrics.

Handling this evolution—particularly at local scales—requires also that we can learn these properties *online*. Pre-trained models to solve a task or implement some control mechanism such as a CCA are likely to be trained in a 'one size fits all' manner from a vast amount of data. This works well in the general case, of course; functions are often trained to handle the most common scenarios and behave well in response. In reality, a characteristic of our deployment environment might not have been included in training; either a useful property like a network's structure, or a problem dynamic like the local protocol distribution. In the event of either this scenario or

some gradual change in the underlying problem, it is reasonable to tailor an existing policy to suit our current needs. Alternatively, we might aim to prevent performance degradation arising from complex dynamics we don't yet know how to model but can see in live networks, needing us to train from scratch. Online learning techniques, such as RL and federated versions of unsupervised and semi-supervised algorithms, allow us to adaptively train policies to consider local tailoring and global evolution. Standard supervised ML approaches, on the other hand, require significant caveats to achieve similar tailoring; labelling data is expensive (either by hand or by constructing and running relevant simulations), datasets are too large to transport, and training datasets may have privacy or ownership concerns attached. RL in particular avoids this by using incremental performance metrics which should be easily observable at runtime. However, online learning adds concrete difficulties beyond simple ML inference. Even complex functions like *Deep Neural Networks* (DNNs) can be made efficient on weaker hardware using specialised representations or data formats, but their training relies on holding high volumes of data in memory, alongside costly procedures for computing the gradient estimates needed to update them.

While these developments enable (online) DDN and network programmability, it is at their intersection that the field of networking is truly on the cusp of something promising. For instance, PDP hardware and the integrated network measurement it enables expose new sources of data and state, such as port and queue occupancies, or precise and accurate flow telemetry. In the case of flows, DDN processes can then act based on such data rather than sampled metrics, potentially offering more accurate classification for uses like QoS assignment. In the network at large, they may act on a network-wide picture of device state which *would have otherwise been unavailable.* Although this data is evidently too much for any one host machine to process—particularly per-packet events at a switch's aggregate $\mathcal{O}$ (Tbit/s)—the enhanced programmability allows PDP devices to pick up the slack by acting on it *in situ.* The first way we could achieve this is by aggregating and reducing data in PDP hardware to make it feasible to export (at lower volumes and rates), or to apply early statistical processing to ease the workload on process machines. The second is by *moving ML and RL logic directly into PDP hardware.* DDN decisions may then be instantly factored into the routing and processing of typical dataplane *Match-Action Tables* (MATs), at minimal latency cost. In concert with online learning techniques such as RL, we may also tailor this behaviour to suit the deployment device or location—either by tweaking a known-good base policy, or learning from scratch. However, programming in-network services has its own challenges: the hardware offers restricted instruction sets, program lengths, data types, *Functional Units* (FUs) for capabilities like floating-point support, and memory. Each limits the kinds of processing we aim to perform, but nowhere is this felt more keenly than the mismatch between the needs of ML algorithms and capabilities of network hardware. This grows greater still with our desire for

online learning in the PDP context, which requires us to compute and represent incremental and optimal changes to a learned policy. While this logic could also be pushed to host machines, there are significant drawbacks in decision latency and throughput. This thesis explores and chronicles how these approaches benefit the network and one another, as well as the challenges introduced by the limited capabilities of a PDP environment. To that end, I show through the community's advances and my own additions to the state of the art how these approaches can offer meaningful benefit to network operators, as well as how we might enable both online and offline learning in the PDP networks of tomorrow.

1.1 Thesis statement

This thesis asserts that:

> Data-driven networking—enhancing networks with ML—and dataplane programmability are key tools in aiding the control and measurement of future networks (*s0*). Data-driven methods such as reinforcement learning can lead to improved performance in network optimisation and control problems, such as DDoS prevention (*s1*). In-network compute can make data-driven networking more efficient, effective, and responsive—enabling online learning to tailor policies to their deployment environment (*s2*). Finally, dataplane programmability will allow the precise measurement *and* data aggregation that can enable fine-grained data-driven analyses to scale to high flow rates or large networks (*s3*). Applied together, programmable data-driven networks can improve computer network operation beyond the sum of these parts.

While claims *s0–1* fall in line with expected uses of these new technologies, the others require some extra explanation to unpack. Claim *s2* may be somewhat surprising, if we think only of the massive $\mathcal{O}$ (MiB–GiB) models which dominate classification, control, and language tasks. By considering changes to algorithms and numerical formats, smaller models can be executed in the limited resources of PDP hardware. The architecture of these devices is specialised around processing high rates of packet events—by parallelism or pipelining—which can allow line-rate operation of models transformed as above. If such decisions can be made *at the same time and location as input data arrive*, then the network can (re-)act faster. This also affords us more time to compute gradients and the like without impacting per-packet behaviour, making online learning feasible. Claim *s3* arises due to the scale and volume of data which PDP hardware can produce. Consider a single 100 Gbit/s port on a switch, operating at line rate with a mean packet size of 500 B, from which we want to make some DDN decision based on PDP-only

state such as ns-level timestamps and queue occupancies. On average, this produces 25 Mpps events *per-port*, which is difficult for a single machine to handle—let alone when it must perform per-packet inference. PDP hardware thus has an important role to play in digesting and summarising its newly available metrics for host machines.

1.2 Contributions

Grouped according to the claims (*s0–3*) in the thesis statement:

- *A thorough summary of the literature on modern, programmable computer networks (including recent hardware trends) (chapter 2), and of machine learning techniques suitable for their control (chapter 3).* This includes the history of a spectrum of tools developed to optimise data-plane processing—both automatically and by bespoke design. Not just how ML benefits networks, but how creative PDP-enhanced networking can benefit ML use cases (*s0,2*).
- *A novel synthesis of best practices, design decisions, and environmental tradeoffs to consider in the design of ML-led system control (section 3.1.6, s1).*
- *An improved RL-based DDoS prevention scheme (chapter 4, s1).* This builds on two protocol-agnostic, flow-granularity RL agent designs (*Instant* and *Guarded* action models), alongside algorithmic modifications to Sarsa to enable better concurrent learning from many in-progress RL trajectories, and reward functions tailored to detecting the negative impacts of amplification DDoS and *Link-Flooding Attacks* (LFAs). This is supported by a quantitative investigation of suitable flow features for attack traffic detection via RL, deadline-aware action planning and state fusion to shield agents from being overloaded, and a concrete architecture and design of a *Virtual Network Function* (VNF)- and SDN-based installation of this anti-DDoS solution. To assess this work, I introduce procedures and trace data for modelling and generating traffic similar to modern Opus-based *Voice over Internet Protocol* (VoIP) flows. This is then used in an empirical evaluation of these models against the prior state-of-the-art in RL-based DDoS mitigation and a non-ML algorithm tailored towards LFAs.
- *OPaL—the first implementation of in-network, online RL (chapter 5, s2).* This includes an analysis of why RL in PDP hardware is needed and best-placed to interact with the network, made feasible by classical RL methods and quantisation. In support, I design an RL interaction model based on path-adjacent compute to protect carried traffic, offer an analysis of suitable data formats for online DDN in resource-constrained hardware. A new proof is given that 1-step temporal-difference RL algorithms admit a parallelisable, map-reduce form with

tile-coded policies, culminating in *ParSa*—a wait-free, parallel, online RL algorithm to accelerate tile-coded policy inference and updates. This allows a design space exploration of parallel RL strategies tailored to provide either maximum offline throughput, or optimal state-action latencies and online throughput, as well as work allocation algorithms and communication tailored to SmartNIC devices with an explicitly tiered memory model. OPaL is evaluated in-depth—how it affects carried dataplane traffic, performs in latency and throughput under different policy sizes (simple and complex state), and improves on host machines. Finally, I describe how OPaL would integrate with state-of-the-art PDP applications to perform fully in-NIC, fast, automated DDoS mitigation.

- *Seiðr histograms for aggregation of precise flow telemetry (chapter 6, §3).* This is a flexible dataplane-assisted architecture and algorithm compatible with the *Portable Switch Architecture* (PSA) that allows data aggregation in the form of histograms. The use of histograms is supported by a measurement study of *Inter-Arrival Time* (IAT) microstructure between *Transmission Control Protocol* (TCP) CCA variants, and analysis which establishes the algorithmic cause for these differences. Using ML methods, I present a high-accuracy method for using the Seiðr procedure to track IATs with nanosecond-accurate timing to tell apart timer-based (e.g., BBR) and *cwnd*-based TCP CCAs using host machines. Its effectiveness is shown by an extensive evaluation of TCP congestion control classification using IAT histograms in different ML models, as well as analysis of Seiðr's scalability compression ratio relative to input sequence length.

A contribution I can't claim to offer, but hope sincerely to have done, is to collect together enough of the literature and intuition on DDN and PDPs to serve as a comfortable introduction to a newer researcher in the field. The topic of DDN in particular has blossomed during the course of my PhD education—and scarcely existed at the scale it does today when this work was first undertaken in 2017. Making the case for its relevance and best practices has become much easier over the last few years alone in light of this. I'm fortunate that the work of many others tackles the same problems as I do, which I think lends credence to the thesis statement (*so* in particular)—in a sense, this work contributes one set of case studies among many. I hope that this thesis can be the book I would have wanted to read (and use) as a starting point when I was setting out on this research venture.

1.3 Thesis outline and structure

Broadly speaking, this thesis is presented in two halves. The first offers in-depth background on both the fields of DDN and PDP:

Chapter 2 describes the evolution of computer networks from fixed-function devices towards increased programmability in both the control plane and dataplane—critically examining early research directions in contrast with modern successes. It then describes how modern dataplanes improve or allow new networked applications—namely, offloading and in-network compute.

Chapter 3 provides an introduction to the new field of data-driven networking by critically reviewing the design of many recent ML solutions to network problems and relevant function approximation and learning methods. This includes data formats needed to run ML techniques in resource-constrained environments, and concludes with some discussion on the limitations and security context of ML.

The second half presents novel, concrete use cases which each demonstrate a part of the thesis statement (as discussed above):

Chapter 4 investigates using multi-agent RL to automatically learn the features of attack traffic online. I explore agent designs informed by past RL approaches (and their failures) relative to the realities of Internet traffic, while discussing the threat landscape of volumetric DDoS attacks. State spaces in particular are experimentally justified to find 'per-feature' value. A system architecture as part of a larger VNF system is shown, followed by evaluation of efficacy on different traffic classes and scenarios.

Chapter 5 takes to task the goal of enabling in-network, online RL for the first-time. I present an exploration of the design space around the interaction mechanisms, compute models, algorithm modifications, and data structures needed for PDP devices. This high-level design is named OPaL. It then presents significant implementation detail for OPaL on *Netronome Flow Processor* (NFP) SmartNIC hardware, followed by performance evaluation to show its improvements in state-action latency and to assert that its impact on traffic is minimal.

Chapter 6 examines how in-network data reduction to histograms can make complex, non-latency-sensitive ML decisions on host machines scalable. I motivate their use with a measurement study on CCA detection from per-flow ns-level timestamps, before evaluating their general scalability and effectiveness in the target use case.

The thesis then concludes by summarising its main takeaways, offering closing thoughts on these fields, and outlines future work specific to the above use cases (chapter 7).

Additional, supplementary details follow:

Appendix A describes the methodology and results of a small-scale study on the distribution of protocols in CAIDA trace data, to establish a rough estimate of congestion-unaware traffic's presence in *Internet Service Provider* (ISP) networks.

Appendix B provides additional detail on the measurement process used to collect trace data for simulating VoIP-like traffic, as well as the software architecture for packet generation.

Appendix C expands on architectural details for the NFP family of SmartNICs to offer some additional context for OPaL's design constraints.

Appendix D contains packet header and protocol descriptions for OPaL's in-band control protocol.

Chapter 2

Programmable Computer Networks

Computer networks serve the important function of allowing any two machines to communicate with one another, typically via individual messages known as packets (i.e., a packet-switched network). Naturally, reality is much more complex than this broad statement would otherwise let on; the local routing fabric in a modern network comprises specialised (though commonplace) hardware for correctly routing these packets through arbitrary topologies of links and switches at ever-increasing data rates. This grows more complicated still when we consider the task of *inter*networking between such networks, where we must route packets on higher-level logically structured addresses between different domains of control according to fairly complex policies and relationships. At the inception of these technologies, computer scientists of the day wisely decided that the sole duty of the network itself should be the correct routing of individual packets. Their view was that application-level logic should be executed solely at endpoint machines; their definition extended, of course, to even include desirable (and some would say indispensable) transport-level properties such as error checking and stream reliability. This is known as the *end-to-end principle* (Saltzer *et al.*, 1984). This position arose partly due to the logical complexity of all the tasks pushed onto the network at this time, as well as the need to ensure optimal forwarding performance while microprocessors were still relatively nascent, but was instrumental in ensuring that the network itself remained *extensible*. A consistent, pared down feature set was ensured while offering a good degree of freedom for the development and deployment of higher-level protocols.

Decades have passed since then, and to a large extent the zeitgeist has shifted on just how capable our networks should be—in both the research community and operators of large-scale networks. Consider the case where an operator has a fully converged network built entirely on fixed-function hardware, but wishes to use some program to inspect the behaviour, state,

and characteristics of some flow between two local machines. The problem is that these devices offer no means of modifying or influencing routing state, being highly-optimised switching devices that understand a selection of routing algorithms built into their internal circuitry. For the longest time, altering the network's routing behaviour in this instance—even for a single override—required not only physically altering and rewiring the network, but also would require additional hardware. SDN was a key development in enabling this fine-grained routing over traffic at various layers in the protocol stack, allowing operators to offer per-flow or per-class routing for improved performance—TE—or even application-aware load balancers at the switch level. Initial forays into SDN were built on exploiting a separate *control plane* to install MATs and rules on target switches—mapping fields of predefined protocols to predefined actions—leaving truly complex decisions to one or more controller machines. These developments have been pushed even further as the runtime capabilities of supporting devices have evolved into what we might now consider truly *Programmable Dataplanes* (PDPs). A wide variety of ASIC-based switches, SmartNICs, and other accelerators now offer an environment for expressing and executing truly arbitrary network logic, protocol parsers, and action definitions.

Despite all this, our general-purpose Internet remains much the same from an endpoint perspective—performance and reliability improvements aside. Yet this increase in capabilities has revealed new strands of research in more specialised networks such as data centres, where in-controller processing would have allowed the network fabric to cooperate with its hosted applications but presented an obvious computational bottleneck. *In-network compute* is enabled by such bespoke routing environments when combined with the above advances in programmability, and is founded on the growing idea that in-path network elements such as switches, NICs, and middleboxes can (and should) host complex logic to accelerate applications, participate in flow control, or to aid in network management.

This chapter begins by motivating and describing initial attempts at dataplane programmability in the '90s (primarily *active networking*), how control-plane programmability and SDN arose in their wake, and the reasons behind these movements' respective failures and successes (section 2.1). Section 2.2 introduces modern, programmable dataplanes by tracing efforts parallel to the development of SDN for improving the performance of host-based packet processing, such as VNFs, before leading into the emergence of specialised PDP hardware from legacy SDN and NPUs. This is followed by commentary on the ways that the original active networking movement differs from modern PDPs (and context behind the latter's successful adoption), as well as a selection of open challenges and proposals in PDP programming languages and hardware designs. Section 2.3 then explains the rationale behind *offloading* service logic to PDP hardware and into host network stacks, while describing recent research on using these capabilities to automatically accelerate existing dataplane programs. Finally, to further

motivate in-network compute I describe point solutions which take advantage of the execution environment and more granular view of network data to improve measurement and operation (section 2.4).

2.1 From fixed-function to software-defined

For historical value, and to provide some context on the design and architectural decisions of modern programmable network stacks (section 2.2), it is important to consider early developments and advances which laid the groundwork for the PDP ecosystem as we know it. This includes the ill-fated active networking movement, leading into the development of control plane programmability via SDN.

Initially, network fabrics were *fixed-function*, supporting only the routing algorithms provided by permanent ASICs integrated with their silicon, and offering transit only for protocols considered at their construction. However, from the Internet's origin as ARPAnet through today, programmability of networks has increased over time to simplify the management, use, and adaptability of network infrastructure (Feamster *et al.*, 2014). Programmability in computer networks tends to be categorised into two distinct forms. *Control plane* programmability focuses on the routing of packets, making it easier to alter, update, and tailor the forwarding behaviour of a network at run time, and at many levels of granularity. *Dataplane* programmability focusses instead on introducing additional logic into the network to be executed by the forwarding elements such as routers—stateless or stateful transformations of packet streams, traffic measurement, and so on. We examine first *active networks*, one of the earliest movements to enable network packet processing at the infrastructure level.

2.1.1 Active networking

In response to the expanding scale and widespread reach of the Internet (and computer networks in general), researchers in the early-to-mid '90s increasingly desired the tools to extend, innovate, and research routing and transport protocols. To maintain and safeguard interoperability over the Internet, the *Internet Engineering Task Force* (IETF) formed to maintain and oversee the development of Internet protocols for all levels of the networking stack. The weight of full IETF standardisation was seen by many researchers of the period as a lengthy process, which they believed to be the cause of *network ossification*—the Internet becoming inflexible to the design and deployment of new protocols.[1]

Active networking was researchers' response: a family of clean-slate proposals built around enabling switches to perform arbitrary computations on carried packets, and for the network to share resources such as com-

[1] Authors of this period might be horrified to discover that the IETF's median time to standard publication has more than doubled since 2000 (McQuistin *et al.*, 2021). This is not, however, what we understand as ossification in today's Internet, which I'll discuss shortly.

pute and memory with users and applications (Calvert, 2006; Tennenhouse & Wetherall, 1996). This is a more communistic, cooperative view of the role of the network—that it should provide and manage advanced services as a sort of common good beyond raw forwarding capacity and functionality, more than simply 'best-effort'. This would enable not only new protocols empowered by the cooperation of the routing fabric, its proponents argued, but would also simplify network management and measurement; it was in no uncertain terms a radical departure for its time, standing in stark opposition to the simplicity demanded by the end-to-end protocol. Active networks planned to enable caching and *Content Delivery Network* (CDN)-like behaviour, stream compression, network management, and enhanced telemetry. Similarly, they could transparently improve data transfer with in-path compression or error correction via *protocol boosters* (Feldmeier *et al.*, 1998).

Surveys of today divide the ideas of this movement into two main streams of research (Feamster *et al.*, 2014):

Capsules, which consisted of compact programs bundled with network packets; either at a per-packet level, or installed on a per-flow basis during the handshake process.

Programmable switches, which allowed arbitrary programs to be installed by system administrators to their own infrastructure for dataplane processing.

These are two key tools rather than opposing schools—and works we'll examine from the tail end of the *active networks* movement feature a high degree of interplay between both ideas. It must be said that in the absence of specialist supporting hardware, the vast majority of works in this field relied entirely upon execution of high-level code via commodity host machines. *User Datagram Protocol* (UDP) tunnelling and overlay networks like *PlanetLab*[2] (Chun *et al.*, 2003) were necessary to do so; the former modelling a cooperative multi-*Autonomous System* (AS) Internet in a testbed setting, and the latter providing hosts with CPU and memory slices across the world in a ticketed, quid-pro-quo manner.

[2] PlanetLab also had a wider effect on distributed systems research. Sadly, it was shutdown in May of 2020 (Peterson, 2020).

Innovations in capsules *ANTS* (Wetherall, 2002; Wetherall *et al.*, 1998) captured the stereotypical active networking idea of 'arbitrary user programs' carried by each packet. A capsule contains one or more program IDs in each header's packet, to be inspected by an ANTS runtime at on-path active nodes. Capsules include dataplane programming and routing logic, kept immutable between flows. An active node checks its local cache for program code matching a capsule's IDs, which may be pre-installed by an administrator (out-of-band); on a cache miss, the code is requested from the last active node (in-band). Programs for a given ID are verified and signed externally by some trusted organisation e.g., the IETF, but their use is still

controlled by *the user or endpoint application.* ANTS relied on the transfer of Java code: extensions such as *PAN* (Nygren *et al.*, 1999) investigated the use of raw unsafe x86 assembly for performant in-kernel use.

Smart packets (Schwartz *et al.*, 2000) examined dataplane programming from the perspective of measurement and control. Intended as a means for management centres to install packet programs on managed nodes, one or more initialisation packets would be sent across the network between a source and destination—each containing a single complete program and all relevant authentication certificates. Any nodes along the path—including endpoints—would be free to install or pass on logic as required. Programs were limited, stateless, compact programs executed in a *Virtual Machine* (VM) for all carried packets, with dedicated intrinsics for accessing state from the *Management Information Base* (MIB); packets could trigger MIB events, or modify a packet to include per-path telemetry supporting passive and active measurement use cases. Hosts and switches were expected to enforce limits on execution time and dynamic memory use to keep the scheme feasible at scale.

NetScript (da Silva *et al.*, 2001) allowed for the definition of active network programs as a dynamic (i.e., potentially branching) dataflow graph of smaller programs—*boxes.* Such boxes may be recursively defined. Crucially, its main advantage over the similar *Click* (Morris *et al.*, 1999) is that any box can be a remote node (another machine, or a specialised hardware/ASIC implementation), which makes this a remarkably prescient combination of control plane and dataplane programmability. Dataplane program definition and selection is left entirely to network operators in this abstraction, rather than allowing tenants' code.

Switch programmability Though conceptually purer uses of programmable switches are rare in this movement, their use is often implicitly assumed to be necessary in any real, performant active network. The *SwitchWare* project (Alexander, Arbaugh, Hicks *et al.*, 1998), while very much a capsule-based proposal, primarily suggested a base of *SANE*-backed switches (Alexander, Arbaugh, Keromytis & Smith, 1998) which would implement a fixed, performant set of 'active extensions' analogous to *Operating System* (OS) syscalls. Ahead-of-time fixed subprograms like these were key here to achieve performance and security—mainly by limiting capsule program capabilities and allowing delegation to ASIC accelerated operations in performance-critical scenarios.

From a more practical perspective, Wolf and Turner (2001) examine the design requirements of a programmable switch in contrast with emerging NPUs of the time. They propose a *System on a Chip* (SoC) design built around a large array of general-purpose *Reduced Instruction Set Computer* (RISC) CPUs, each having their own cache and memory. A first stage ASIC would route packets to an output port over a shared interconnect, where

these CPUs would either count as their own port or be attached to a physical egress and conditionally used.[3] The use of complete RISC CPUs is quite deliberate: early NPUs such as the Intel IXP1200 (Intel, 2001) and Lucent's Fast Packet Processor (Agere Systems, 2001) had unusual instruction sets suited to stateless processing of headers (George & Blume, 2003), limiting their general expressiveness. A fuller *Instruction Set Architecture* (ISA) was recognised as being necessary for more capable, general, useful, and stateful dataplane programming; particularly of the kind envisioned by active networking.

[3] Curiously, this design more closely matches modern SmartNIC devices, while today's programmable switches favour ASIC designs. We'll return to this point in section 2.2.2.

Returning to *smart packets*, we see a similarly earnest attempt to reconcile capsule networks with a reasonable programming framework (as opposed to its peers' focus on high-level languages). Aiming to be more amenable to switches they propose *Spanner*, a compact *Complex Instruction Set Computer* (CISC) assembly language for a stack-based VM with primitives to access the MIB, compiled to from their own higher-level language. Spanner was designed to be reasonably implemented on hosts and routers, but explicitly trades performance for compactness; focussing on small, self-contained, secure programs.[4] The executing switch or host enforces memory and execution limits at runtime, but single-packet $\mathcal{O}(1\,\text{KiB})$ programs simplify deployment logic. Naturally, this excludes stateful boosters of the type we've examined, but is well-suited to the measurement and management use cases its authors intended.

[4] This shares many conceptual similarities to *Extended Berkeley Packet Filter* (eBPF), a RISC register machine, which now plays a key role in network compute offload and OS kernel extensibility. I introduce eBPF in detail during section 2.3.1. The VM abstraction, using intrinsics and maps to communicate with the environment, ensures security while the language's simplicity allows implementation in the network fabric.

Failings and drawbacks Sadly, the active networking paradigm failed to properly fledge, at least in its first iteration. It might be argued that the 'communistic' network view played a key part in this downfall—that the tangled web of ASes between any two nodes should freely offer compute resources to the benefit hosts. This comes down to a simple question of economics: who pays whom for providing these services? It's easy to reason about this in campus or internal networks (the organisation provides active capabilities as part of its own remit), or if we require that only our ISP offers these capabilities (hosts pay for them explicitly). Extending this notion to the wider Internet becomes more challenging. Assigning responsibility for administrative, technical, and security issues among all the organisations between two endpoints is a daunting prospect, to say the least. When modifying packets in a protocol booster-like model, it becomes difficult to communicate where boundaries of support start and end to enable truly transparent behaviour. In a setup like the PlanetLab overlay network the incentives for providing these capabilities in such a distributed way are obvious: all the users are researchers in need of distributed compute. Each benefits from donating compute and network slices in their own infrastructure to receive resources in kind.

The problem here is also one of aggregation. At the Internet core such as in *Internet eXchange Points* (IXPs), transit bandwidth demands are the sum of

all connected ASes, amplifying compute demands and scale concerns. From another angle, suppose that all functions benefit the network *and* hosts alike, as in the case of an in-band TCP compressor. Here, files can be transmitted between end hosts quicker *and* the bandwidth impact on the switching fabric is reduced. If hosts benefit and have the resources to implement this functionality it is inevitable that at some point they will do so (pushing logic to the network edge), at which point network operators are able to put their own resources into faster or higher-capacity passive infrastructure with a vastly reduced attack surface compared to an active solution. On the upside, this also preserves the end-to-end principle.

Active network capabilities do in many senses *limit* how networks can benefit their users (or at least make some innovations substantially harder). Consider flow migration, potentially as part of a wider TE strategy, or path aggregation to increase bisectional bandwidth or provide redundancy to protect from link failures. Administrators must not only handle routing and design of such capabilities, but they must also ensure that active network programs on any flow's path are mirrored or migrated to the new path. In the event of a complete or partial node failure, this may not be trivially possible. Even when retrieving programs from the last hop, failover may be needed for a live high-bandwidth flow, requiring costly per-flow buffering at a node until capsule program setup is complete.

The capabilities of programmable switch hardware were, to some extent, overshadowed by the community's focus on the more intoxicating idea of capsule networking. Naturally, the overwhelming majority of these platforms were prototyped on host machines, which limited forwarding and processing performance far below that of even early Ethernet. The repeated emphasis on high-level prototypes reduced the focus on the sorts of low-level, capable languages a performant system would require—proposals were instead marred by in-vogue high-level VM-based languages such as Java and Caml, which did their credibility little good. This likely also added to industry scepticism—one gets the sense of a paradigm driven by research trends instead of limiting its own boundaries to produce something truly capable and scalable.

2.1.2 Software-defined networking

In parallel, a good many researchers saw a need to innovate (and fight stagnation in) the space of what we now understand as control plane programmability: the ability to deploy and develop new routing protocols, provide bespoke routing for individual packets, flows and applications, and to adapt to changes in the network itself. While likely starting out from the same perspective as active networking—enabling new capabilities and development, and network evolution—as years have gone by greater network capacities, usage demands and a need for far greater reliability have introduced chal-

lenges manyfold. Greater capacities, particularly in bisection bandwidth, typically mean more routers and switches for network admins to manage, but also in terms of additional cabling for link redundancy and aggregation—neither of which play well with older spanning tree protocols. *Traffic Engineering* (TE)—making effective use of this capacity and providing different traffic classes with optimal forwarding—has similarly become more and more important over time. The problem is that classical routing protocols such as *Open Shortest Path First* (OSPF) and *Routing Information Protocol* (RIP) are designed for distributed autonomy, offering no support for direct routing rule insertion by administrators. Early TE approaches were thus built on a shaky foundation of hacks and tricks, exploiting routing protocol behaviour to achieve the desired high-level outcomes (Feamster *et al.*, 2014). Naturally, such workarounds become untenable and impossible to reason about at scale—particularly as failures and misconfigurations creep in and become harder to find and diagnose. Effective though these early methods were, they remained hamstrung by the tightly coupled control and data planes of the network hardware of that era. The reality and urgency of this problem space has led to the true separation of forwarding elements (the dataplane) from the logical control elements which inform and orchestrate their routing behaviour (the control plane). With the advent of the control plane, *Software-Defined Networking* (SDN) arose too—the use of arbitrary machines and logic to define the forwarding behaviour of each and every network element (Feamster *et al.*, 2014; Nunes *et al.*, 2014). As we shall discuss, this has brought forth a well-developed and successful bevy of innovations in network control and design.

Development of the control plane This line of work began in earnest with the *open signalling* movement (Campbell *et al.*, 1999) in *Asynchronous Transfer Mode* (ATM) networks, circa 1995. Open signalling aimed to standardise the *Application Programming Interfaces* (APIs) and protocols for passing routing data to its relevant handler (and rules to line-cards) in an era where these elements were still tightly bonded in commercial routing hardware. *Tempest* (van der Merwe *et al.*, 1998) marked its zenith, using these capabilities in tandem with specific functions of these now-obsolete ATM networks to define a 'network control architecture' which would oversee correct forwarding policies and resource allocation. Its primary focus, however, was in providing independent virtual network slices—*switchlets*—to users of multi-tenant networks on demand, allowing these users to pass in their own control programs written for host execution in Java, for instance. Control plane traffic from control elements to the dataplane was carried *in-band* at this time (i.e., using the same fabric as tenants' datagrams).

These ideas were refined into a standard for hardware elements on a shared bus via *ForCES* (Yang *et al.*, 2004); stripped, of course, of the more visionary aspects such as network slicing, arbitrary control code, and wider resource management. Primarily, this was focussed on registration of control

plane elements, including key provisions for extensibility and discoverability to announce additional protocols and offer redundancy. The move to physically remote control elements was proposed by *SoftRouter* (Lakshman *et al.*, 2004), such that forwarding and control elements would communicate with one another in-band over the same network they define (after a simple spanning-tree bootstrap phase) using custom discoverability protocols. This allowed its authors to experiment with reducing the number of control elements in a network, from which they observed faster convergence of protocols such as OSPF. Hardware capabilities did not at this point allow this level of control, and as such host-based kernel dataplane routers like *XORP* (Handley *et al.*, 2003) and simulators such as *ns* (Bajaj *et al.*, 1999) reigned in research at this time.

Most of the above work focusses on the *intra*-domain case—routing *within* an AS. *RCP* (Caesar *et al.*, 2005; Feamster *et al.*, 2004) was instead motivated by how *inter*-domain routing *between* ASes might benefit from the sorts of centralised control we now associate with SDN. The problem in this instance is that edge routers must understand and implement *Border Gateway Protocol* (BGP), each having their own arcane policies to achieve the intended network behaviour with respect to quirks of the protocol, their own hardware, and the routes forwarded by neighbouring ASes. Their solution was to centralise this logic into one (or a few) host machines, providing a drop-in solution which could act on *all* gathered BGP information to produce optimal decisions with better scaling characteristics, reduce administrative overhead and fragility, and which opened the door to later extension if neighbour ASes also deployed RCP.

With the growing need for TE and *Traffic Optimisation* (TO) at various layers of the Internet infrastructure, we begin to see a drive for wider coordination in the intra-domain case. *PCE* (Vasseur *et al.*, 2006) is one of the first serious attempts to formalise a network architecture where the control plane can calculate and install bespoke *paths* at the flow level, given any set of constraints. Naturally, this requires far greater coordination of the individual forwarding behaviour of switches, preferably combined with *Multiprotocol Label Switching* (MPLS). *Ethane* (Casado *et al.*, 2007) pushed this notion of complete network control further; a dedicated controller overseeing the forwarding rules and paths installed at an entire network of 'dumb' switches, containing *only* an action table of *exact* matches. Their goal was not one of performant TE, however, but of policy—complete control over interactions between named entities and their classes, with all flow misses being directed to the controller.[5] The controller then authenticates, authorises and orchestrates all host-to-host connections according to a given policy.

The arrival of *OpenFlow* (McKeown *et al.*, 2008) marked a watershed moment in SDN, iterating on Ethane's key ideas to remove notions of authentication, focussing more on the network itself. Motivated to lower the barrier of entry for experimentation and research in the routing domain,[6] Open-

[5] It's worth noting that this proposal was intended only for campus networks, hence its obvious scalability limits don't bite quite as badly as they might in, say, an IXP or data centre. In particular, it must upcall to the controller for every new address, and maintain an exact match rule for every live flow due to the lack of longest-prefix matches.

[6] In-text, this is again referred to as ossification: here arising from the closed nature of routing hardware coupled with the performance needs of realistic traffic.

Flow decided to change tack compared with XORP and its contemporaries by targeting real hardware (and by tailoring its design to capabilities of commodity devices of the day). OpenFlow kept Ethane's MAT abstraction, as switches of that era stored flow tables internally, and required support for few actions: forward packets on a port, drop packets, encapsulate and send a packet to a designated controller, or defer to the switch's underlying forwarding logic. Each OpenFlow switch then maintains a dedicated, secure connection to at least one controller, though may receive rules from any number of them. This was set apart from its earlier competitors and progenitors by its drive for easy compatibility with commodity hardware, for instance by providing their own open source OpenWRT firmware. What it then codified was an open, consistent manner for software to configure the routing behaviour of real hardware at the whim of operators, going so far as to suggest that high-performance dataplane programmability might be enabled via steering to NetFPGA devices elsewhere in the network—eliminating virtualisation. The supported capabilities and reach have expanded somewhat since its initial, academic, limited introduction. Indeed, while we now understand that OpenFlow has incredible value in the operation and management of large networks, the vision embodied by the original work is in fact much closer to the 'switchlet' model of Tempest where individuals request transit for traffic between their machines, carrying their own experimental protocols or enacting novel policies. The specification has thus evolved over time to include features more useful and tailored to complex networks and costlier hardware: conditional support for more expressive actions and matching capabilities, including additional tables, groups, egress processing, and packet header modification (Open Networking Foundation, 2015).

An ongoing legacy The remarkable thing about the field of SDN, particularly compared to active networks, is that it's still here. Moreso than that, it plays a large role in modern networks of hypergiant scale. We'll discuss a case study shortly. Why has control plane programmability proven to be so much more effective and attractive to operators in practice?

Principally, it is because control plane programming is easier to get right—in the sense of processing speed and volume. Routing information, link states, and topology changes collectively arrive at a far slower rate than packets, or even simply carried flows. An OpenFlow controller is under no obligation or need to, say, react to millions of routing changes per second while adding only microseconds of latency, because the routing *environment* varies at a rate which is well suited to the processing capabilities of commodity hosts.

Secondly, the wide reach of open-source software for developing, deploying, and testing SDN concepts and ideas has helped these ideas gain traction in both the research community and in real-world deployment. Network OSes and controllers such as *NOX* (Gude *et al.*, 2008), *Onix* (Koponen *et al.*, 2010),

ONOS (Open Networking Foundation, 2021), *OpenDaylight* (OpenDaylight Project, 2021), and *Ryu* (Ryu SDN Framework Community, 2017) provide moderately easy interfaces for programmers to implement their own controllers. Network emulators such as *mininet* (Lantz *et al.*, 2010) allow researchers to define and operate virtual networks built entirely of OpenFlow-capable switches on their own machines using host programs and standard sockets to generate and use carried traffic, enabling easy prototyping. High-performance software switches such as *Open vSwitch* (OVS) (Pfaff *et al.*, 2015) offer first-class support for OpenFlow and no small number of extensions, and are instrumental in data centre virtualisation (Tu *et al.*, 2021).

Finally, the movement capitalised on *existing hardware* and its own capabilities, rather than calling for a radical wave of enhancements or brand new, expensive capabilities. Providing a higher-level abstraction to manage the infrastructure which networks already contained ensured the adoption and current success of SDN.

One well-documented, well-published, and arguably ongoing case study of this paradigm in action at scale is the wave of high-impact papers produced by Google. Google's *Jupiter* data centre network design (Singh *et al.*, 2015) was developed concurrently with most of the works outlined above, where these techniques have been instrumental in managing the explosive uptick in switch hardware demanded by their larger Clos topologies.[7] Additionally, SDN techniques have helped to manage phased integration of this new network. In *Wide Area Network* (WAN) topologies, *B4* (Jain *et al.*, 2013) has made direct use of OpenFlow for TE, to maximise link utilisation and offer resilience (aided by the fact that Google own the WAN endpoints). The architecture evolved fairly cleanly over the following 5 years, coping with 100 × bandwidth increases and more stringent reliability bounds (Hong *et al.*, 2018). *Espresso* (Yap *et al.*, 2017) extends this to inter-domain peering arrangements with the Internet and BGP session management, finally realising the vision of RCP 13 years on. Nowadays, their data centre and WAN networks are managed by a single higher-level microservice SDN architecture, *Orion* (Ferguson *et al.*, 2021).

[7] Particular issues demanding custom routing and administration include the vast amount of multipath complexity at this scale, heavy per-link state, and potential performance gains from network homogeneity that general-purpose routing algorithms might fail to capture.

2.2 Modern programmable dataplanes

In parallel with SDN's development, dataplane programmability has become more commonplace and performant—primarily through a mixture of novel hardware architectures and more developed software stacks on host machines. These tools offer a useful mixture of capabilities, performance characteristics, and drawbacks to consider. To better understand how best to use these tools in service of DDN—for building full systems built on device-local state (chapter 6) or the primitives for inference and online learning (chapter 5)—this section details current hardware and software dataplane architectures and their predecessors. This also gives us cause to examine

the wider context of these recent PDP developments, and why they have succeeded as compared with past research forays.

While active network research may have dried up in the last 20 years,[8] interest in dataplane processing has remained strong. Network operators have, in fact, always had a need for complex packet processing to measure, protect, and enhance *their own* networks—rather than to provide the sort of communal good that active networks promised. To offer some examples:

- *Firewalls* are a necessity for filtering unauthorised traffic at stub ASes, but require line-rate lookups versus large allow- and block-lists across varied protocols. Similarly, *Network Address Translation* (NAT) boxes play an important role for ISP networks.
- Protocol enhancements and TE solutions, such as *WAN optimisers* and *application-layer load balancers*, which need to be able to inspect and/or modify traffic based on (possibly complex) transport-layer and application-layer semantics.
- Security functions, such as *Intrusion Detection Systems* (IDSes) and DDoS traffic scrubbing solutions. These can require complex, stateful logic (such as regular expression matching and transparent datagram reassembly) that makes them challenging to implement while maintaining high performance—particularly when we consider that the DDoS use case *demands* that we keep up with a high ingress traffic volume.

At scale, the performance challenges involved in per-packet and per-flow processing for these functions becomes difficult to reconcile with the performance limitations of host machines. The obvious solution is to fall back to silicon in pursuit of performance, and as a result the market has long included so-called *middleboxes*—bespoke ASIC devices designed to perform one or more pre-defined dataplane functions at line rate.

An obvious trade-off has been made here: runtime programmability has been sacrificed for performance. What is less obvious is that another, hidden, cost has been introduced owing to the fallibility of engineers—our final source of network ossification. While effective at their designed tasks, middleboxes are infamous for relying on the *observed* behaviour of network traffic rather than the behaviours actually decreed in the relevant IETF standards and specifications. As a result, they have become a barrier to the wider deployment and introduction of protocols in the Internet. *Stream Control Transmission Protocol* (SCTP) (Stewart, 2007) is one recent example of an approved protocol whose deployment has been hobbled by the prevalence of middleboxes that expect only a limited suite of layer-4 protocols (Xin, 2021). Under the same considerations, the *QUIC* transport protocol (Langley *et al.*, 2017) must be tunnelled over UDP, even though its Internet-wide deployment is backed by the weight of hypergiant provider Google.

[8] Barring some very obvious throwback works such as *Tiny Packet Programs* (Jeyakumar *et al.*, 2014)—which have directly influenced new in-network telemetry schemes like those we cover in section 2.4.1.

These devices also introduce operational and logistical challenges around their configuration and installation within the network. Although SDN-based steering has to some extent obviated the need to physically rewire middleboxes' cabling when they need to be reconfigured, they do still present challenges beyond the above network fragility. Specialised middleboxes are expensive, demand rack space and introduce their own power and cooling demands, making deployment harder to scale out as network traffic demands increase. They are failure-prone, require specialist knowledge to operate, and are vulnerable to vendor lock-in (Sherry *et al.*, 2012). Moreover, their fixed nature makes them difficult to modify, upgrade, and fix; a sunk cost which cannot be recouped or repurposed as network function requirements change or evolve.

The result of these drawbacks is that a constant desire has remained to introduce and capitalise on true dataplane programmability. The pursuit of this goal can be divided quite neatly into two streams. The first carries on from the use of virtualisation—already common in active networking research before the introduction of a true programmable dataplane fabric—towards more flexible deployment through *Network Functions Virtualisation* (NFV) and VNFs. The second stream instead follows from asking how we might make switching and forwarding hardware itself more programmable, keeping in mind the tight form-factor and performance bounds required of high-speed commercial network hardware.

2.2.1 Virtualisation and commodity machines

Commodity CPUs are already arbitrarily programmable, yet their dataplane performance has always been at odds with ever-improving Ethernet standards. In turn, researchers have asked: how can we alleviate the existing performance bottlenecks in host dataplanes? How can we take further advantage of the flexibility of host machines to allow for multitenancy, or make runtime reconfiguration even easier by leaning into the ubiquity of host compute? Backed by SDN and by innovations in VM and container research from the systems community, VNFs and similar frameworks have filled a comfortable niche for moderately performant host dataplane programming.

NFV and VNFs envisioned that *Network Functions* (NFs) should be implemented as commodity software programs running within VMs, allowing not only the above increases in asset reuse and programmability but also enabling hardware hetereogeneity, portability of developed functions, and resilience (Chiosi *et al.*, 2012). Casting dataplane programming in this light has its fair share of advantages. Administrators can always scale horizontally to meet traffic processing demands by acquiring more commodity host machines, and the dataplane functions themselves are easy for typical software engineers to program—no proprietary languages or hardware details

need to be involved. Bespoke development is not required either; VMs may easily run openly available, widely used IDS software like Snort (Roesch, 1999; Snort Team, 2017) or Zeek (Paxson, 1998; The Zeek Project, 2020) with minimal effort. Moreover, VNF setup requires on the order of minutes or less, and VNF execution can be dynamically stopped, restarted, or moved around; capitalising on the strengths of virtualisation (Cziva *et al.*, 2015; Martins *et al.*, 2014). The VM model then allows for easy multi-tenancy and resource sharing, governed by a hypervisor *à la Xen* (Barham *et al.*, 2003) or *KVM* (Kivity *et al.*, 2007).

Naturally, widespread adoption of container frameworks[9] has percolated into the design of VNF deployment strategies. Container VNF solutions, such as *GNF* (Cziva & Pezaros, 2017; Cziva *et al.*, 2015), have taken advantage of containers' lightweight virtualisation to offer marked improvements in traffic processing latency and throughput over their priors. Their reconfigurability, combined with clever use of network resources empowered by SDN, has also opened the door for *orchestration* research which maximises VNF performance subject to environmental limits. VNFs and the steering between them may be optimised ahead-of-time (or live) to protect users' application latency or QoS, by *Integer Linear Programming* (ILP) models (Cziva *et al.*, 2018), heuristics (Iordache-Şică *et al.*, 2021), or data-driven methods (Riera *et al.*, 2016).

[9] *Containers* are a form of OS-level virtualisation, with a history reaching back to *FreeBSD Jails* (Kamp & Watson, 2000). These are a lighter-weight alternative to full VMs, having a single shared OS which provides filesystem isolation and virtualises access to I/O devices. While isolation guarantees are weaker around NF performance, this does enable some runtime benefits, such as fast container-container network communication on the same node.

While the above works relate mainly to the higher-level composition and interconnection of network functions, a long lineage of works drawn from *Click* (Morris *et al.*, 1999) have instead focussed on how to build these packet programs from smaller chunks for efficient processing and easier development. Click itself views the dataplane as a configurable graph of interconnected, predefined building blocks which are then composed into a single, pipelined program. *ClickOS* (Martins *et al.*, 2014) introduces a micro-OS designed only to run Click programs as part of a Xen hypervisor stack, achieving substantial packet rate and spool-up time improvements, while *Fastclick* (Barbette *et al.*, 2015) innovates in performance by porting Click on Linux hosts to use Intel's *Data Plane Development Kit* (DPDK). *ClickNF* (Gallo & Laufer, 2018) then introduces support for higher-level stateful protocols such as the TCP stack, enabling *Hypertext Transfer Protocol* (HTTP) caches and proxies as network functions. *NetBricks* (Panda *et al.*, 2016) maintains the directed graph model and DPDK support, however it takes user-written functions rather than pre-specified blocks, and relies upon the *Rust* (The Rust Team, 2022) language compiler to enforce isolation of memory and packet data accesses. *OpenBox* (Bremler-Barr *et al.*, 2016) makes use of a similar framework of smaller configurable processing blocks, though its main focus lies in fusing NF components in larger chains across several machines. When functions are co-hosted on any machine, OpenBox merges their processing graphs to deduplicate more expensive operations (reducing processing latency) while allowing some degree of metadata transfer between hardware-optimised functions and commodity hosts.

We have also seen specialisation of these frameworks to account for security at varied levels, due to the importance of security functions such as IDSes and the proliferation of VNF chains deployed to remote cloud compute. For instance, VNF operation must be protected from other VNFs and the host machine it runs on (which may be compromised), as must packet data and proprietary VNF source code. *SafeBricks* (Poddar *et al.*, 2018) extends NetBricks to take advantage of *Trusted Execution Environments* (TEEs).[10] This executes as a single binary in the TEE, with another thread running in userland to operate DPDK as a dedicated packet relay due to the prohibitive cost of swaps in or out of the trusted enclave. Such swaps cause substantial delay via complete cache flushes and re-encryption. *AuditBox* adds verified routing protocols to this framework (G. Liu *et al.*, 2021), enabling packet routing *between enclaved VNF chains* on separate machines—ensuring also that packet paths are obeyed, and that the network is not altering or reordering packets. Additionally, this allows path auditing in a way which is invisible to VNFs and the network. However, this latter solution achieves only half the packet forwarding goodput of NetBricks in exchange for these additional guarantees.

[10] TEEs allow secure execution of code on a target machine to be isolated from the kernel and all other running processes, typically by using encrypted memory pages accessible only to the CPU. Platforms such as Intel SGX also include provisions for binary and machine attestation.

2.2.2 Specialised hardware

Although host-run functions have indeed improved, their performance still trails unacceptably behind ASICs—traffic volume can be accounted for up to a limit by horizontal scaling, but this becomes more difficult to operate than a single bump-in the wire solution, or may be impossible at larger port densities. Programmable network hardware, beginning with early NPUs and culminating in today's SmartNICs and programmable ASIC-backed switches, innovates by introducing runtime reconfigurable compute to these devices.

NIC programmability NPUs, dedicated network cards including general-purpose processors for arbitrary packet processing, first arrived around the year 2000 as we've discussed in section 2.1.1, with commercial designs such as the Intel IXP1200 (Intel, 2001). These presented an early middleground between fixed-function (though performant) ASICs and commodity host (virtualised) packet processing, offering line-rate performance at a smaller form and fanout factor (i.e., 1 or 2 ports). By 2008, their architectural space was dominated by multiprocessor SoC-type designs offered by a good many vendors (Giladi, 2008, p. 306).[11] Yet at this time there was still a high degree of microarchitectural diversity between vendors (Keutzer *et al.*, 2002; Shah & Keutzer, 2002)—pipelined versus symmetric core designs, *Very Long Instruction Word* (VLIW) and superscalar alongside one-way designs, and the presence or absence of use case-specific FUs and coprocessors. Some architectural constants include a large, shared register file to enable zero-cost context switches and mask I/O latencies, and non-trivial programming models. These devices fell behind ASICs in deployment and use for the longest

[11] Following up the fates of these providers paints an interesting picture of monopolisation by traditional network vendors: consider for instance Sandburst's acquisition by Broadcom (Informa Tech, 2006), or EZchip's acquisition by Mellanox (Cohen, 2015).

time due to the speed and low fabrication cost of chips—even if their design was costly (Giladi, 2008, p. 308)—twinned with the need for deep expertise needed to program such NPUs.

SmartNICs present the logical evolution of this low port-density form factor, offering line-rate packet processing for modern Ethernet (10–100 Gbit/s). What sets them apart is that SmartNICs optionally function as a NIC compatible with common OSes—supporting for instance the standard offloads such as checksumming and segmentation. Modern day SmartNICs such as the NVIDIA/Mellanox BlueField (NVIDIA, 2021b) and Netronome Agilio (Netronome, 2021) series trace their lineage back to these NPUs, to some extent realising the architecture which some authors thought programmable switches might adopt (Wolf & Turner, 2001). That is to say, large numbers of weaker *general-purpose* RISC CPUs which can route to one another or an output *Medium Access Control* (MAC) via a shared interconnect, with ample coprocessors (for, e.g., cryptography) and resources for accelerated packet processing like *Ternary Content-Addressable Memory* (TCAM). This contrasts the network-specific ISAs used in their predecessors,[12] while acting as a usable NIC for the host machine in addition to their bump-in-the-wire capabilities. Packet throughput requirements are met by sheer parallelism in these devices—Netronome's NFP-6480 contains 112 cores acting in a non-pipelined way (appendix C)—but such NPU-type SmartNICs often impose greater latency on carried packets due to architectural overhead.

[12] Mellanox NICs in particular use standard ARM cores, while Netronome's offerings use a proprietary ISA with full C language compatibility.

Field-Programmable Gate Arrays (FPGAs) have become more capable over time, and now present a reasonable way to achieve ASIC-like performance (high throughput *and* low latency) and runtime reconfigurability without fabrication. The advent of the openly available NetFPGA (Lockwood *et al.*, 2007) design as a platform for network processing research and teaching has been a key driver here, such that the SmartNIC market also includes FPGA-backed NICs such as the Xilinx Alveo (Xilinx, 2021a). Similarly, other vendors have produced open NIC designs, such as AMD/Xilinx's *OpenNIC* (Xilinx, 2021b) platform. NetFPGA processing speeds are also continuously improved in tandem with line rate: see NetFPGA-SUME (Zilberman *et al.*, 2014) at 40 Gbit/s, and NetFPGA-PLUS (Tokusashi, 2021) at 100 Gbit/s.

Switch programmability Let us turn now to programmable *switches*, noting that the path they have taken is altogether different from what Wolf and Turner proposed some 21 years ago. It is readily apparent that the designs we seen in modern SmartNICs would be broadly incompatible with high port densities (i.e., 56 ports) as the sum traffic bandwidth rises to 1–2 Tbit/s and beyond. Operating and connecting together hundreds or thousands of general purpose cores requires vast amounts of power and cooling, to say nothing of the engineering challenge of designing an efficient interconnect for all these FUs. We quickly reach a point at which ASICs are the only

means to achieve some manner of programmability; we must instead ask *how much* freedom to give programmers, and then architect around this restricted programmability.

Serious contenders began to reach the market in 2013, such as Intel's *FlexPipe* (Intel Networking Division, 2017) and Cavium's *XPliant* (Cavium, 2017). However, due to its publication and later impact we will leave these aside in favour of RMT (Bosshart *et al.*, 2013). RMT's motivation followed on from the design and limitations of OpenFlow, asking not only how MATs could be made faster in hardware but also how a new architecture might dynamically allow for novel protocols and encapsulations such as VXLAN. To do this, the switch must offer runtime reconfiguration of field definitions and locations, the shapes of MATs, actions themselves, and control over output queue selection and disciplines. The RMT model divides the switch into ingress and egress pipelines with a queue and buffer management element between them, with an array of configurable TCAM-backed parsers to operate on the inbound packets in each pipeline. Pipelines are subdivided into 32 stages, each having 200 action units that act in parallel on the packet header vector using a VLIW instruction chosen by a MAT lookup. Because of this, each stage can perform a single operation per field, though the control plane can make use of the large number of action units to perform speculative execution and divide and split logical stages between parts of the physical pipeline. This design sees wide adoption today. Barefoot—now Intel—*Tofino* switches use a switching core built on this architecture, the *Protocol Independent Switch Architecture* (PISA) (Barefoot, 2017).

The question which then remained was how the control plane, and engineers in general, should actually program these advanced classes of switching hardware to exploit these new features. *Protocol-oblivious forwarding* (H. Song, 2013) was an early proposal in this regard which failed to gain substantial traction; a single 'flow instruction set' would act as a common compile-time target to govern protocol parsing logic and action implementations, while even allowing in-band modification of MAT entries by actions. In retrospect, its main failing is that it overlooked how ergonomic, high-level programming should be carried out while also advocating a hardware redesign not entirely backed by a sensible implementation. *P4* (Bosshart *et al.*, 2014) is a language which instead targets the design of a dataplane—its table layouts, actions, and parser design—from a high, user-friendly level. Users program parsers as a *Finite State Machine* (FSM), while table and action sets are structured as imperative programs based on the classes of match offered by a target switch. Switches are supported by defining their architecture in headers and implementing a new backend for the compiler, *p4c*. Actions have access to *metadata* (per-packet state) and *registers* (shared dataplane state), where the latter is analogous to OpenFlow's *counters*. P4 supports many target dataplanes today:

- Programmable switches, through either vendor-provided backends

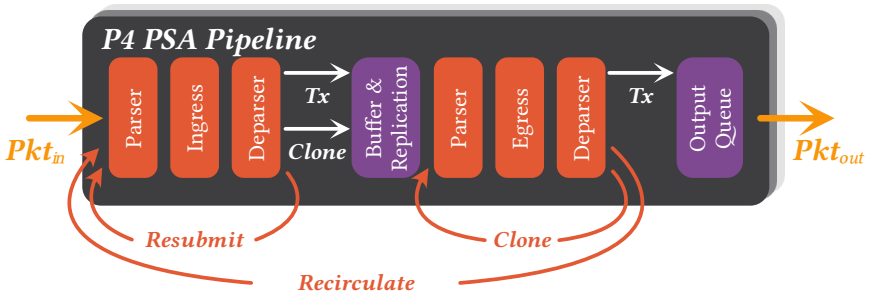


Figure 2.1: Pipeline stages and forwarding paths of the P4 PSA. User programmable blocks are coloured in orange, where MAT blocks comprise the 'Ingress' and 'Egress' stages. While an RMT-like action model is common, the PSA abstracts over how actions should be implemented in any target dataplane. Instead, it specifically determines the high-level structure of ingress and egress processing—as two separate parse-MAT-deparse pipelines—and how packets may be moved and cloned between these pipelines. For instance, a packet may only be cloned at the beginning of the egress pipeline, and may only repeat processing by returning to the start of ingress. Due to this structure, some packet processing or aggregation programs may only be expressed using workarounds.

such as the *Tofino Native Architecture* or shared targets such as the PSA (The P4.org Architecture Working Group, 2021)—a higher-level abstraction over Barefoot's PISA. Figure 2.1 demonstrates the PSA in particular.

- eBPF (P4c Maintainers, 2017) targets (kernel-level, userland and offloads),[13]
- SmartNICs, via vendor-specific compiler additions,
- NetFPGA cards, via the P4→NetFPGA pipeline (Ibanez, Brebner *et al.*, 2019),
- Software switches such as *bmv2* (Bas, 2016), *PISCES* (Shahbaz *et al.*, 2016) built on OVS, and the more performant *T4P4S* (Vörös *et al.*, 2018) which compiles to C code with a DPDK backend.

[13] P4 and eBPF each have semantics the other cannot express, which are given in detail in the referenced documentation, so this translation is less than perfect for mapping some P4 dataplanes to relevant offloads.

Any device intrinsics or capabilities which cannot be expressed in P4 may be called out to using `extern` directives. Generating compatible control plane protocols and APIs is then handled by *P4Runtime* (The P4.org API Working Group, 2021), restoring the hardware-agnostic control central to OpenFlow.

8 years on, P4 has apparently been a runaway success in much the same vein as OpenFlow, and enjoys use in production networks (Tian *et al.*, 2021) while also enabling efficient and production-ready implementations of next-generation Internet designs such as *SCION* (de Ruiter & Schutijser, 2021). These technologies, including SmartNICs, are also seeing use in national research and education networks—for control and network design such as

Géant's RARE project (Meyer, 2020), and for fine-grained network telemetry and future P4 processing in ESnet6 (Guok *et al.*, 2021; Mah *et al.*, 2020).

A more important effect of having arbitrary control over actions and manipulation of shared state is that, although limited, the set of programs we can express has grown significantly. Although we'll discuss this in more depth in section 2.3, this enables true *in-network compute*—per-packet processing, e.g., in telemetry use cases like *PINT* (Basat *et al.*, 2020) at rates and volumes hitherto impossible for hosts to achieve. Moreover, this enables application acceleration and other new developments with none of the costs of host execution.

2.2.3 The return of active networking?

The reader may well be thinking that research community has cycled back around to active networking in a new guise, and in many senses it has. While less popular at the time, the programmable switch model we have now latched onto did in fact arise as part of this prior movement. The remarkable observation is that if we follow the retelling of Feamster *et al.* (2014), active networking's decline was written in large part by its lack of a "killer app". Of course, its main use cases at the time were touted as enabling cacheing and CDN-like behaviour, content processing, network management, and fine-grained telemetry. These are strikingly similar to the use cases which have driven the recent upsurge of PDP applications, and in fact see real use in production edge networks (Tian *et al.*, 2021). Modern measurement schemes like *In-band Network Telemetry* (INT) (The P4.org Applications Working Group, 2020)—covered in section 2.4.1—have returned to the model of injecting per-packet actions into the packet headers themselves.

Wetherall and Tennenhouse (2019) also recognise this resurgence, and reaffirm the main drivers of SDN—*economics* of a malleable software layer exploiting affordable commodity hardware (where talented software engineers are easily deployed), and *virtualisation* as a tool for introducing more capability into the network. What they don't really discuss are the concrete reasons for why the field appears to have successfully taken off this time, while it foundered before. So, what changed? My personal interpretation and opinion is that this arises from several angles:

- Virtualisation introduced the capability to install novel and reconfigurable packet processing to networks, but modern deployments and the constant need to 'scale up' have emphasised that performance truly is a necessity. This doesn't only hark back to the logical cost involved in moving packets around the software or OS stack in a machine (added latency), but also via the falloff of Moore's law (G. E. Moore, 1965) and Dennard scaling (Dennard *et al.*, 1974). Although host capabilities have always fallen behind line-rate, they are falling

even further back due to the constant, inexorable increase in Ethernet data rates.[14] 'Line-rate' has increased from its initial 10 Mbit/s to 1 Gbit/s (Frazier, 1998), to 100 Gbit/s (D'Ambrosia, 2010), with ongoing work to standardise 400 Gbit/s links. When we consider that Ethernet frame sizes have in the same period expanded on *some* deployments from 1560 B to 9000 B 'jumbo' frames, it is plainly visible that per-packet processing deadlines simply cannot be met by commodity hosts on smaller packets.

[14] Of course, dedicated silicon always offers a performance edge over an arbitrary computer, at the cost of flexibility. The design, fabrication, and engineering costs are high enough that this has only ever really been justified in common use-cases (e.g., firewalls and IDSes) or where there really is a business case as with modern hypergiant providers.

- The unforeseen capabilities, reach and business needs of hypergiant network operators such as Google and Meta, pushing the boundaries of scalability (Gigis *et al.*, 2021), have also played a key role. Crucially, their business needs include not only the administration of such large networks but also rely on accelerating cloud compute workloads, large-scale ML model training and inference, and distributed computation within their datacentre networks. As we'll cover in section 2.3, in-network compute allows specific optimisations for these tasks: for instance gradient aggregation in-network, or CCAs managed co-operatively by the routing infrastructure. In single-owner environments such as these, all aspects of distributed computation can be controlled and optimised, so such hyper-converged infrastructure is not only possible but necessary from an economic standpoint—particularly when in-network compute becomes the only available road to greater performance. While such companies have the capability and precedent to develop their own hardware—e.g., network-connected accelerators like Microsoft *BrainWave* (Fowers *et al.*, 2018)—these organisations already had an abundance of engineers familiar with SDN who were poised to make great use of the joint flexibility and performance offered by PDPs.

Alternatively, we might argue that it is the control plane innovations of SDN that made this possible in the first place; beforehand, the design schism of capsules versus programmable switches was indeed an open question (one might say between 'pragmatic' and 'interesting' approaches to the task). The field's re-evolution of programmable switches (and now NICs) offers a healthy does of pragmatism, ensuring that today's model is performant—but it is almost entirely focussed on allowing runtime reconfigurable specificity over which packets are fed to a pre-set menu of dataplane programs, as opposed to totally arbitrary packet-level programs.

Yet the model of in-network compute we have converged on remains radically different from early estimates, even though it may feel like the community is simply retreading the concept of 'programmable switches'. Consider the motivation behind *SmartPackets*:

> There are places, however, where Moore's Law is winning. One place is network management and monitoring. The average

> device is not generating, processing, or receiving drastically more network management traffic than it was a year or two ago. We can hope, therefore, that there is more per-device processing power available for network management than there was in the past.
>
> (Schwartz *et al.*, 2000, p. 68)

At that time too, Moore's law was insufficient to allow per-packet processing at cutting-edge Ethernet speeds (before its falloff really came to pass). As such, the only time that *full, general-purpose compute* could be deployed in this context was when the infrastructure had already aggregated or reduced the data frequency in some way. The authors here make two key, and arguably fatal, assumptions. The first hinges on a perceived status quo: that management and telemetry data would *and must* remain low-frequency (e.g., flow-level, link-level, or sampled measurements). Section 2.3 shows that for many applications this cannot be the case, as per-packet processing and telemetry are essential in accelerating distinct use cases and diagnosing insidious network faults and behaviours such as microbursts. The second is that the compute model itself should be equivalent to host machines, able to express any packet processing programs an engineer might dream of. We see this in other programmable switch proposals of the era (Wolf & Turner, 2001), but time has shown that NPU-like SmartNICs of today can only achieve this for one or two ports—let alone the full fanout of a rack-mounted switch. The first key difference lies in how, at scale, we must make use of advanced (though highly programmable) ASICs rather than full-fledged CPUs. While the design of SmartNICs allows us to dispel the first assumption and enables exciting new use cases, in the switch form factor we have come to accept *constrained*, yet still capable, programmability to meet line-rate processing. The other key difference between the modern PDP ecosystem and active networks is that the scope of deployment has narrowed considerably. While early proponents dreamed of a fully participative network, inclusive of end-hosts' in-network programs, PDP devices must be programmed ahead-of-time and managed by an attached controller machine—both for performance and for management of the associated control-plane machinery such as MAT structure. In turn, deployment of in-network compute has become far more insular, and effectively bound to the AS level.

2.2.4 Frontiers in programmable networks

Commercial developments along the same lines as this modern PDP hardware are proceeding apace as network bandwidth demands grow larger. Intel's Tofino 2 (Intel, 2022) represents the latest product in the lineage of RMT hardware, offering 12.8 Tbit/s with support for 400 Gbit/s Ethernet. Nokia's FP5 (Nokia, 2021) similarly promises full programmability for high-density

switching and routing at 800 Gbit/s Ethernet, while Intel's *infrastructure processing units* (Intel, 2021b) present a combined FPGA- and Xeon-based series of SmartNICs for accelerating datacentre applications. However, there is still concerted research effort in further developing the tooling used to program these devices and in how future hardware designs might evolve to incorporate new models of packet processing.

Language design At present, P4 and the PSA are restrictive in the sense that the only events which can drive user-provided code are packet arrivals and departures. Event-driven languages have been suggested (Ibanez, Antichi *et al.*, 2019), built on the need for timer, link state, and queue state events to enable useful applications. Workarounds in P4 exist to emulate these capabilities, such as queue size estimation and costly packet recirculations, but these inflate the amount of state needed by applications or incur their own runtime penalities. The authors explicitly incorporate these events into the pipeline model and modify the P4 language to support their processing by additional logical pipelines; however, this demands hardware support in non-NetFPGA environments. *Lucid* (Sonchack *et al.*, 2021) builds a new, high-level language which expresses many of these capabilities by compiling down to the P4-PSA architecture. In particular, it allows for event handlers to be triggered between devices while enabling more flexible control and modification of shared datastructures behind reliability measures like fast reroute.

The P4 language is currently incompatible with heterogeneous hardware to some extent; written dataplane programs are typically tied to a particular switch model, for instance V1Model, PSA, NetFPGA SUME, or the Tofino Native Architecture. Network architects typically procure hardware from several manufacturers to prevent vendor lock-in, but the P4 model for each device has its own metadata types, hardware constraints, and quirks which developers must be aware of. While architectures such as the PSA are more general and should, in principle, support several target switches, it is often preferable to use a switch's own architecture for performance or optimisation reasons. As a result it is currently tedious to write and maintain a unified dataplane that is provably uniform across different packet processing devices. *μP4* (Soni *et al.*, 2020) extends the P4 compiler to decompose parser and action code into independent subprograms which may be composed together in a more simple manner by programmers. This simplifies porting behaviours between different switch models—particularly in separating out (and integrating) complex interactions and dependencies between parser, deparser and action code stages. *Lyra* (Gao *et al.*, 2020) is a language for running switch programs e.g., P4 and Broadcom's *NPL* (NPL, 2019), across heterogeneous switch hardware, while also handling placement constraints. Lyra expresses the entire network dataplane using the 'one big switch' abstraction, and compiles from its own higher-level language to an *Intermediate Representation* (IR) and then to P4 or NPL. Compilation is combined

with topology information about the target network, as well as placement constraints, to generate an optimal embedding in the network using a *Satisfiability Modulo Theories* (SMT) solver.

As is the case when programming host machines, verifying that a dataplane program behaves correctly—both within a single switch and the wider dataplane with regards to code and MAT contents—presents its own set of challenges. In particular, fully programmable dataplanes enable new classes of bugs such as header malformations which existing network verification tools are not designed to handle. This is complicated further still by the fact that a P4 program's operation is determined also by the control plane and the contents of its MATs. *P4-NoD* (McKeown *et al.*, 2016) is an earlier solution to the problem, translating invariants into Datalog for verification by older tooling while modelling correctly emitted packets via pairwise 'packet acceptance' constraints between switches. *p4v* (J. Liu *et al.*, 2018) combines guarantees about the bounds of control plane values with expected output invariants to detect counter-example packets using an SMT solver. *bf4* (Dumitrescu *et al.*, 2020) endeavours to make the annotation task considerably easier for programmers, again relying on SMT solvers to also produce control plane rule filters and candidate bug fixes. *Aquila* (Tian *et al.*, 2021) achieves a similar class of SMT-solver based verification, using a new language which makes it easier to express dataplane invariants across one or more switches. Aquila further innovates by using SMT counterexamples to localise likely locations and fixes for bugs, as well as developing numerous domain-specific optimisations to generated logical formulae.

Hardware design While the pipelined model of the PSA is undeniably effective, it can be restrictive for many classes of dataplane program; for instance cases where processing is based on more than raw packet events, or where more complex dataflow is required between functions. *PANIC* (Stephens *et al.*, 2018) offers one solution by placing a routing fabric between distinct packet and data processing elements in a SmartNIC. These compute elements (mixed RMTs, FPGA blocks and accelerators) are connected in a tiled architecture, each containing a router to direct packets to their intended internal destination. Such a design would enable general, asynchronous, and novel compute in SmartNICs and switches, for instance offering consistent and easy to use communication between workers versus hard-coded *Microengine* (ME) relationships or ordering dependencies between subprograms across pipelines.

In multitenant environments such as data centres or cloud compute providers, clients may wish to take advantage of PDP hardware if it is present—between pairs of virtual servers, for instance. Recalling the single pipeline design of the PSA, it's clear that this is a difficult resource sharing problem between MATs, pipeline width and stages, and per-packet metadata storage. Moreover, ensuring that applications cannot interfere with one another's

performance guarantees, state, or the forwarding behaviour of all packets is non-trivial (i.e., a malicious table might force infinite packet recirculation)—particularly when tables or logic might be reused between user pipelines to save such resources. Alternative architectures have been presented to make this task simpler. The above *PANIC* has been revised and recast as a solution to this multitenancy problem (Lin *et al.*, 2020), losing much of the flexibility of its initial iteration to accelerate this use case. (Multitenant) PANIC now uses a single ingress RMT pipeline to tag packets with all hops of their intended offload chain, while compute units (ASICs and RISC-V CPUs) pass packets between one another using an all-to-all direct crossbar. Higher-level rate limits are controlled by a programmable *Push-In First-Out* (PIFO) scheduler (Sivaraman, Subramanian *et al.*, 2016) to enforce QoS around shared offload blocks. However, relying on only the ingress RMT to determine such routes leads to potentially inflexible packet processing chains. *MTPSA* (Stoyanov & Zilberman, 2020) instead extends the PSA to place client code into a set of inner 'user' pipelines between the egress parser and MATs. Each runs its own code, and applies Unix's read-write-execute privilege model to resources, fields, tables and `extern`s to limit per-program access capabilities. The standard ingress and egress pipelines are designated as 'super-users', who determine the user pipeline to execute and are responsible for higher-level forwarding. This approach is rather coarse-grained, and prohibits reuse of tables (increasing per-user resource costs). In addition, restricting pipeline placement to egress-only limits program expressibility—operations such as changing the output port are illegal for user code.

While P4 registers enable useful stateful programming, concurrent access semantics and pipeline ordering restrictions can make some applications difficult to express. Equally, their implementation in any platform relies on platform-specific `extern`s according to the PSA, and as such their semantics and correct operation will vary on a target by target basis. Architectures such as *Banzai* (Sivaraman, Cheung *et al.*, 2016) and its accompanying C-like language *Domino* compile to MATs internally from restricted *packet transactions*. They differ from RMT by having each action unit (or *atom*) additionally contain a memory unit for shared state, which only it may access (rather than the global, shared registers of P4). Atoms contain a variety of *Arithmetic Logic Unit* (ALU) blocks to enable 1-cycle updates and reads on branches as required for safe concurrent state processing. Domino's restrictions are close to those of eBPF programs, with the extra limitation that only one entry may be accessed per array; the compiler is responsible for building and allocating MATs and concurrently sequencing all transactions. This explicit focus on 1–2 cycle logic blocks allows Banzai to guarantee line-rate execution. *FlowBlaze* (Pontarelli *et al.*, 2019) targets instead *per-flow* state for L2–L4 traffic, mixing MAT blocks with custom extended FSM units. These extended FSMs store variables, can read and modify global registers, and use MATs to look up simple state transition functions based on

fields of all accessible state. States are stored on a per-flow basis, allocated from a hardware-backed hash table. While MATs and FSM blocks may be interposed freely to express more varied programs, true flexibility is only possible at present when these blocks may be easily replaced (i.e., in an FPGA environment). This flexibility has a sharp downside; the variable and state model forces pipeline stalls to clear up hazards around concurrent FSM state accesses, leading to packet drops and sub-line-rate packet processing for some dataplane functions.

Although ML can be made feasible in PDP hardware as I show in section 2.4.4, achieving more complex or higher-precision inference at line rate can only be enabled by dedicated architectural support. *Taurus* (Swamy *et al.*, 2020; 2022) is a proposal to add compute and memory units to the PSA as part of a map-reduce block specifically designed to optimise per-packet ML inference. The proposal has the ingress RMT pipeline now perform feature extraction from packet header data among its standard duties. In particular, it demonstrates that efficient line-rate inference can work using a *Coarse-Grained Reconfigurable Array* (CGRA) of map-reduce units between the ingress and egress pipelines—implementing *Neural Networks* (NNs), *Long Short-Term Memory* (LSTM) networks, or *Support Vector Machines* (SVMs) which process every packet header vector. This CGRA implements a large grid of replicated fixed-point compute and memory units, allowing higher throughput by what the authors describe as 'spatial *Single Instruction Multiple Data* (SIMD)'. This achieves line-rate throughput while reducing latencies from the $\mathcal{O}$(ms) CPU and GPU inference to $\mathcal{O}(10^2\,\text{ns})$, dependent on the target application. In turn, the outputs of any classifiers in this block become available to later pipeline stages, which are able to act upon packets accordingly. Training ML models online using Taurus requires that input packets are sampled alongside local signals such as per-flow QoS metrics to be directed towards a cooperating host machine in the control plane, and cannot be performed unassisted (i.e., purely on-device).

Access to the host programming stack and its full feature set remains an attractive prospect, even subject to the costs discussed throughout section 2.3. NIC-CPU co-designs present a more exotic solution to the latter problem, deeply integrating these two elements together in stark contrast to the typical 'peripheral' view of the NIC. Primarily motivated by optimising around the growing prevalence of µs-level *Remote Procedure Calls* (RPCs) in data centres, *NeBuLa* (Sutherland *et al.*, 2020) eliminates the *PCI Express* (PCIe) interconnect between the NIC and CPU, placing received packets directly into the L1 cache of a target CPU core. This relies mainly upon exclusively using a connectionless RPC transport which enforces fail-fast behaviour for requests which will miss their deadline, allowing shared buffers for *all connections* to be shrunk to fit into L3 cache. The integration of NIC-to-core steering with the L3 cache then allows correct routing to the relevant L1 cache slot for the target CPU core. Base packet forwarding times are thus reduced below 100 ns. *nanoPU* (Ibanez *et al.*, 2021) takes this concept fur-

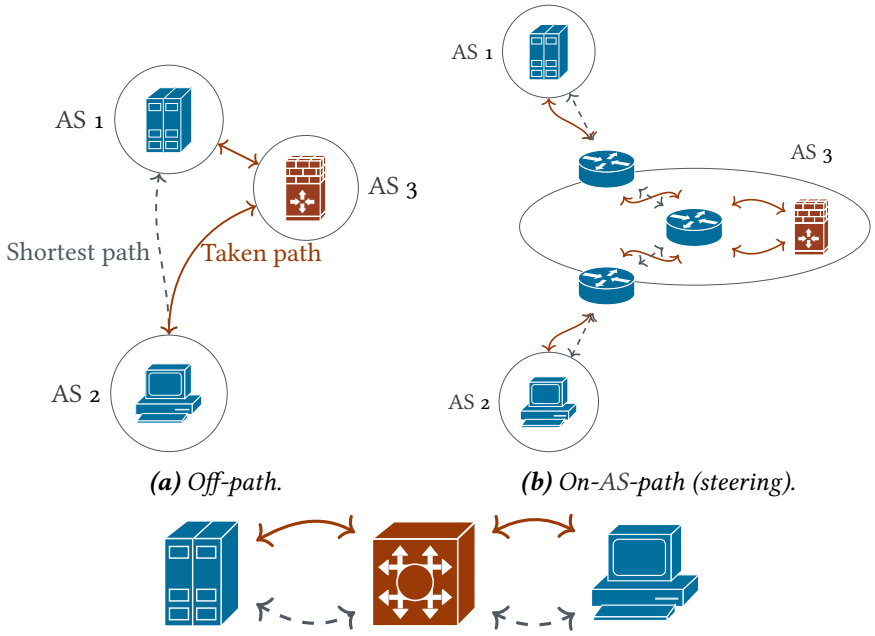
ther to serve packets directly into the register file of a target core using a thread-safe interface, reducing packet handover times to a minimal 69 ns *Round-Trip Time* (RTT) (17 ns excluding MAC). To enable this, custom transport protocol logic is implemented in hardware, and packets are classified and further modified at line rate with the aid of PSA ingress and egress pipelines.

While the P4 PSA is a compilation target supported on many SmartNICs, in many ways it is a suboptimal model as it ignores the constraints and strengths of NIC hardware compared to RMT switches. The *Portable NIC Architecture* (The P4 Language Consortium, 2021), which is in the process of being codified, offers first-class support for such devices, using instead a single pipeline aided by `externs` for host↔NIC processing. This is expected to feature a message processing block for programmable segmentation and coalescing etc., as expected by host OSes and their drivers. Most notably, this includes device-local updates and additions to table state, packet mirroring, and table row expiry events. To date, device-local updates have been implemented as part of T4P4S to explore some of the design-space around concurrent table access and modifications (Simon *et al.*, 2021). However, this capability is not guaranteed for all similar devices. Netronome NFP NIC tables are reliant on the optimised DCFL (Taylor & Turner, 2005) data format, potentially still limiting the capability for rule installation to control plane devices.

2.3 Offloading and in-network compute

As we've explored in sections 2.1 and 2.2, modern networks now have a large variety of tools to enable traffic and packet processing, including hosts, programmable switches, SmartNIC devices, and middleboxes—all in tandem with control plane programmability. This diverse set of devices also means that market silicon now offers a wide variety in performance characteristics such as latency and throughput, connectivity, price points, and degrees of programmability. Accordingly, network architects must also consider how best to integrate NFs backed by these capabilities into their networks. Consider fig. 2.2: to minimise extra latency costs imposed on carried traffic as well as control plane complexity, we want to minimise the amount of *traffic steering* required. In the worst case, traffic may need to be routed to another organisation to make use of high-volume services like DDoS traffic scrubbing, CDNs, or PDP capabilities exposed by cloud compute[15] (fig. 2.2a). Steering within ASes as in fig. 2.2b is a necessity to enable VNF chaining between host machines in a way which is reliably reconfigurable, but this naturally increases routing and processing latency as well as bandwidth demands in the AS in question. The only way to eliminate steering costs completely is to place dataplane programs *on-path* (fig. 2.2c)—but this has key drawbacks. If we place commodity hosts in-line like this, then they

[15] These classes of service tend to have enough geographical replication that the network function and the target server are reasonably close to one another, for instance in proximity to a shared IXP. As such, users' paths to both nodes are likely to be similar, the last few hops excepted.



(a) Off-path. *(b) On-AS-path (steering).*

(c) On-path (direct).

Figure 2.2: To introduce packet processing to the network, engineers must make a conscious decision about where such processing may be installed, and if needed how traffic can be steered there—this leads to spectrum of how on-path processing may be. The main sizes and types of redirections are shown here: (a) having to redirect to another network for packet processing, (b) internally rerouting and steering packets to reach one or more processing machines (e.g., in VNF scenarios), and (c) placing processing *directly in-path*. Generally speaking, smaller path deviations have a smaller latency impact. Paths and AS relationships here are purely for demonstration, and may be longer in practice (similarly, case (b) may occur entirely within a single data centre AS). Appliances performing packet processing are coloured in orange-red.

will be unable to meet line-rate demands in faster networks; we have also given up the flexible reconfiguration and horizontal scalability that steering bought us. PDP innovations like programmable ASIC-backed switches and SmartNICs are the best tools for performing processing here, but not every application can be run in these locations due to the limits of their respective compute models.

These new, specialised devices are a double-edged sword—making efficient use of *all* network resources becomes non-trivial due to device heterogeneity in architecture and capabilities. Ideally, we want to maximise the number of packets which are served by on-path, in-network functions such as in fig. 2.2c. Even when using consistent programming models like P4, using all of a device's capabilities is difficult, requiring hardware-specific expertise, experience, and microarchitectural knowledge. At the same time, organisations would prefer to accelerate the code they already have rather than re-architect solutions from the ground up. Can we then accelerate *individual*

parts of a packet processing stack?

Offloading is the process of moving part or all of a packet processing function elsewhere to improve overall performance—reducing latency or increasing throughput—typically by taking advantage of novel heterogeneous hardware. Originally, this referred to simpler transport-level accelerations that NICs could perform in hardware which would free up CPU cycles on a host machine, such as checksum offloads, large send & receive offloads to split or coalesce larger than *Maximum Transmission Unit* (MTU)-size packets, and receive-side scaling. The idea is that the unique capabilities of existing SDN switches and PDP hardware can be taken advantage of to optimise a target dataplane, enabling a very generic and useful kind of acceleration. This extends also to host machines, which have no shortage of technologies for improving packet processing by shifting user code into the network stack or skipping the kernel entirely. Such hosts may even use attached SmartNICs to accelerate applications or transport logic.

For argument's sake suppose that we want to process packets using a firewall, followed by a primitive statistical or ML model, and followed again by a *Deep Packet Inspection* (DPI) block whenever the second stage emits a warning. Naturally, the simplest deployment is to have all three functions installed on host machines, but this is also the most wasteful. While I'll introduce specific examples later, the only function here which would currently require a host machine would be the final DPI block; even then, this is only required by a small subset of packets which trip an earlier-stage alarm. As such, routing all packets through a VNF chain imposes steering and the higher latency costs of host-based execution upon all packets, to say nothing of the reduced throughput per box. A more optimal solution is to install the firewall rules into the TCAM-backed MATs of commodity switches, which *can* perform these checks at line-rate, and to offload the statistical logic into similarly on-path programmable switches or NICs at a lower arithmetic precision. In this case we have a hardware-accelerated fast path without any unneeded steering, while only packets which need to fall back to a more capable or accurate computation environment take the (less likely) slow path.

The key problem is that automatically offloading arbitrary functions is an open research challenge. Different devices expose different programming models and languages, and have unique capabilities and limitations; additional or missing FUs, code store size limits, bespoke threading models, and other resources. PDP devices have additional constraints on reconfiguration: firmware installation can take from seconds to minutes, limiting a chain's pliability as nodes cease to function for extended periods of time without ample provisioning to handle transition states. Individual functions are also tricky to interconnect (i.e., passing variable state between NF stages), to compute the ideal layout of in the network, and to provision in even single tenant scenarios.

In-network compute is a concept connected to offloading, exploring novel applications which can be enabled or made scalable entirely through PDP hardware. Rather than moving subprograms and arbitrary logical snippets down the stack, in-network compute seeks to move dedicated, complex, or involved applications onto PDP hardware, asking how to take advantage of intrinsic capabilities of dataplane devices. Such research demands more careful exploration in algorithms and data formats, pushing the limits of what programs may be expressed in restricted environments such as P4. What makes this an exciting area of study is that in-network applications are often defined by a tangled net of benefits and costs. Suppose in fig. 2.2c the two endpoint machines first check or update state using another service (e.g., a key-value store) before communicating with one another. By moving this service *completely into the PDP infrastructure*, entire RTTs worth of communication delay can be eliminated. In data centre networks, where RTTs are already $\mathcal{O}$(µs), simply placing a host in-line would undo most of this latency reduction[16] while being unable to meet rack-scale fan-out. This becomes key when dealing with RPC workloads common to such data centres, where completions and RTTs are on an $\mathcal{O}$(µs) timescale themselves (Barroso *et al.*, 2017; Kalia *et al.*, 2019; Sutherland *et al.*, 2020). Further example services may also aggregate data from many sources to allow a single host to process it, or apply per-packet ML inference at line rate; when end-hosts and the network fabric are all jointly owned, there is great scope for tighter network-application integration and what it might enable. Notably, handcrafted in-network services such as ML inference might replace entire blocks in an NF chain, better supporting VNF offloading by making clever use of the underlying hardware. The costs incurred by such services are also interesting. In-network ML must sacrifice accuracy—the lack of *Floating-Point Units* (FPUs) in network hardware forces implementers to employ fixed-point arithmetic or other data formats. Not all programs may be moved down to PDP hardware unaltered.

[16] While host offloading frameworks have come a long way, there are data transfer costs which cannot be elided which we'll discuss in the sequel.

We'll cover here technologies in use for host offloading, such as *eXpress Data Path* (XDP), as well as the rationale and costs of host processing (section 2.3.1). Through section 2.3.2, I'll cover recent works making use of host offloading techniques and PDP hardware to provide automatic acceleration and for dataplane programs. This expands on the raw tooling introduced throughout section 2.2 with a higher level of abstraction, and allows us to better understand how we can push general program logic on-path (as part of a full data and telemetry processing pipeline). It provides us also with some insight into the limitations of bespoke designs and automatic offloading. In-network compute's use cases will be left until the sequel (section 2.4).

2.3.1 Host offloading technologies

Commodity host machines are designed in such a way that packet processing often incurs higher latency than using an ASIC positioned at the

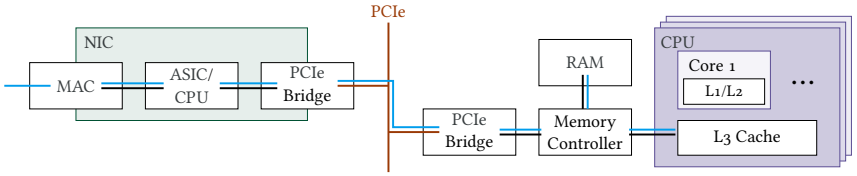


Figure 2.3: A simplified view of the physical packet path on host machines, shown in blue. Moving packets between the NIC and CPU is more involved than simple steering, as packets must be moved across the PCIe bus and DMA'd into host memory. The host CPU is made aware of packet arrivals by either *Interrupt Requests* (IRQs) or polling ring buffers in memory which the NIC writes into. This introduces additional latency, even when the networking stack supports direct insertion into the CPU cache. Note that this figure leaves aside *Non-Uniform Memory Access* (NUMA) constraints and costs arising from having several CPUs.

same point in the network. Consider fig. 2.3, which shows the physical interconnect between a NIC and CPU. Network connectivity is a *peripheral* function, and so NICs are connected over the PCIe bus. To be processed by host machines, the NIC must move packet data across the PCIe bus by *Direct Memory Access* (DMA) into *Random Access Memory* (RAM), where the host CPU(s) can make use of this data—this may be accelerated by copying the data also into L3 cache in the same step, via functionality like Intel DDIO (Intel, 2017). While PCIe offers extraordinary bandwidth (63.015 GB/s in PCIe 5.0), moving data across the bus adds $\mathcal{O}$ (µs) of latency. Quoting figures from Neugebauer *et al.* (2018), 64–1500 B packets spend 0.8–1.8 µs solely in PCIe, comprising 90.6–77.2 % of the total one-way delay (0.883–2.331 µs)—dwarfing the latency contribution of the NIC. These physical costs cannot be removed with standard NICs: the host CPU must have the packet body entirely resident in its own memory to act upon it.

Most of the impact on traffic processing originates also from logical costs due to the OS's device and network stack management. Figure 2.4 lays out some of these stages in the Linux environment. Primarily, the OS kernel is notified that packet DMAs have completed via IRQs, at which point a kernel thread is awoken and the device driver is called to transfer the packet contents into a *Socket Buffer* (SKB) usable by the stack.[17] Awaking a thread does not guarantee that it will be instantly ready to serve the packet, adding latency, while readying packet SKBs adds additional per-packet overheads which harm receive-side latency and throughput. The Linux network stack itself must then inspect SKBs to handle decapsulation, transport logic such as TCP and CCA management, and connection handling—among other functions related in more much detail by existing work (Cai *et al.*, 2021). Finally, the packet is served over a socket to a (possibly sleeping) userland thread, who may require a context switch before the received data may be finally used—again, another source of processing latency.

Interestingly, these additional hardware and software costs are analogous to

[17] The conventional network stack does not poll for packets. While this would reduce any additional delays associated with the interrupt model, running a device driver in a busy loop is not generally considered feasible or acceptable. This model *does* drive specialist frameworks like DPDK, which we'll cover, but this requires care and significant changes in userland code.

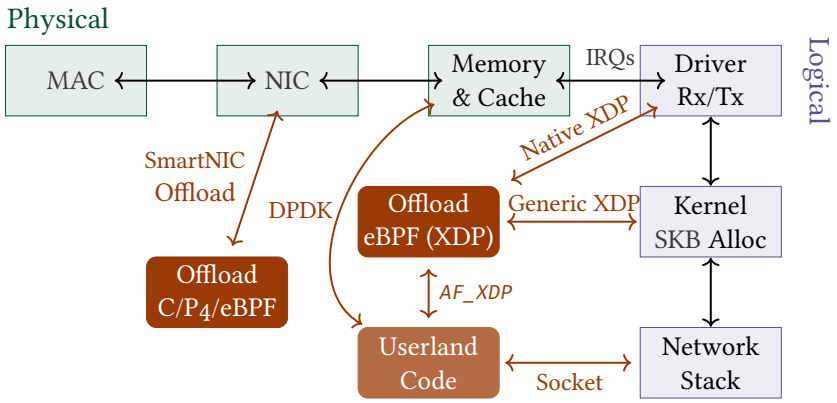


Figure 2.4: The logical packet processing stack on host machines, and how the DPDK and XDP frameworks interface with it and user code. Rounded, filled boxes represent user code. Offload frameworks are useful because they either allow OS kernel code to be bypassed altogether (DPDK), or for packet modification and transmission to be pushed further down the stack (XDP). Offloaded eBPF user code may typically pass packets to the network stack after some amount of processing, send packets directly back to the NIC for transmission, or pass packets to user code using a zero-copy mechanism such as `AF_XDP`. Crucially, all these mechanisms excise various amounts of processing or imprecise waiting for interrupts, reducing latencies and increasing packet processing throughput. More details on the logical portion of the stack are presented by Cai *et al.* (2021).

route-level steering on a smaller scale. A more specialised packet processing stack might be one way to remove many of the software costs, but such a clean-slate proposition would lock packet processing programs out of access to software reliant on typical OS functionality. *Host offloading frameworks* aim instead to reduce this steering as far as possible with support from the OS or hardware; either by eliminating much of the logical packet processing invoked by the OS kernel, or moving user code to an earlier point in the stack. Returning to fig. 2.4, we now focus on the user-programmable blocks. In the best case, SmartNIC hardware allows user code to be moved onto the NIC completely, removing PCIe bus transfer latency as packet processing no longer needs to touch the CPU. Of course SmartNICs are typically less capable than hosts (in clock speeds and included FUs), and for that reason we'll briefly discuss kernel bypass methods such as DPDK and offloads enabled by eBPF. As before, offloaded user code may be some or all of a larger program, potentially divided into fast and slow paths according to whether host compute is needed.

Early optimised network stacks Although less relevant in today's landscape, we'll discuss here some older frameworks offering varying degrees of network stack bypass for completeness. `PF_RING` (Deri, 2004) was motivated by changes in the kernel to prevent IRQ livelock at high speeds, which helped but were insufficient to achieve line rate processing at the receiver. It cre-

ated a kernel-user ring buffer per socket, where received packets are copied into all ring buffers with remaining space allocated to the NIC, bypassing the network stack. *Netmap* (Rizzo, 2012) made use of shared kernel and user memory over ring buffers to place userland code between the NIC driver and the host networking stack for a given interface. I.e., a packet bound from the NIC to the network stack would traverse kernel→user→kernel. It offered specific innovations in copy elimination, batching of syscalls, and effective preallocation of packet buffers and metadata versus SKB-based packet buffer management. Specifically, netmap's rings contain memory descriptors which point into a shared buffer, as compared with `PF_RING`'s explicit byte buffers (which must be copied into).

Kernel bypass In constrast to the above, Intel's *Data Plane Development Kit* (DPDK) (DPDK Project, 2022; The Linux Foundation, 2020) bypasses the kernel entirely, by running NIC drivers in userland. NICs are run via poll-mode drivers in DPDK's *environment abstraction layer*, which manages core mappings to receive and transmit queues, memory allocation and device operation. User programs interface with the abstraction layer to receive packets using a poll-only model, which results of course in consistently high (or maxed out) CPU utilisation. In exchange for this trade-off, programs designed to receive and process packets from DPDK entirely bypass the kernel, greatly reducing per-packet overhead. Since packets must be received from DPDK's abstraction layer, user programs must be rewritten to account for the poll-based semantic model and vastly different buffer lifetime semantics versus the traditional stack—e.g., correct handling and disposing of ring buffer descriptors. This can be worked around to some extent by efficient user-space *library kernel* OS implementations of the traditional Linux networking stack (Thalheim *et al.*, 2021).

eBPF and XDP The *Extended Berkeley Packet Filter* (eBPF) (Fleming, 2017) is a register-based RISC VM and assembly language. Owing to its simple and easily-implemented design, eBPF is used today for moving packet programs early into the kernel, instrumenting standard kernel functions using tracepoints, and offloading. eBPF is derived from the earlier *BSD Packet Filter* (McCanne & Jacobson, 1993), which was a two-register VM designed to allow user-written programs to be safely executed at the kernel level—in this case, to prevent unnecessary packet copies for unwanted traffic or packet contents in monitoring applications like *tcpdump*. 'Arbitrary, sandboxed user code in the kernel' was certainly an idea with potential, and in light of its wider uptake eBPF was modified to become more capable and closer in architecture to modern CPUs: for instance ten 64 bit registers, atomic instructions, maps, and function call opcodes to make use of functions exposed by the kernel.

Today the VM abstraction enables fast and safe execution of such programs

by *Just-in-Time* (JIT) compilation and verification, and is now well-suited for offloading to SmartNICs and similar devices. Making this more accessible, industry-standard compilers support eBPF as a compile target from languages such as C and Rust. In the Linux OS kernel, eBPF programs may be triggered by *hooks* for instrumenting its operation via *kprobes* and *tracepoints*—a program specifies its type with respect to its intended hook, from which the kernel knows which functions an eBPF program may call (effectively enforcing an API). Programs are accepted if and only if they are loop-free, terminate, have bounded size, and simulated bounds checks on array accesses are well-defined. Larger programs may be constructed by building chains of tail calls between these smaller eBPF blocks. Userland and eBPF programs communicate using *maps*, which are a generic abstraction around containers such as arrays, hash tables, and longest-prefix match tables which may be concurrently read and modified by user and kernel code.

Linux's *eXpress Data Path* (XDP) (Høiland-Jørgensen *et al.*, 2018) uses eBPF to place user-specified code into the packet processing path. Naturally, these programs undergo the same verification and JIT compilation as traditional eBPF hook programs, and run one instance per NIC receive queue.[18] XDP hook programs have a slightly more privileged role, and are called before the network stack for each packet arriving on an interface to determine its ultimate fate. For instance, packets may be modified and immediately transmitted back on the wire, dropped, passed on to the remainder of the host networking stack, or redirected to another XDP program or user-land socket. Consider once more fig. 2.4. If driver support is offered, offloaded code may be run before any SKB management or creation with zero memory copies (*Native*). Otherwise, the program runs after SKB creation (*Generic*), but before any part of the remainder of the networking stack. Note that packets may be served *entirely* in the XDP hook by making use of maps, packet modification, and the `XDP_TX` immediate transmit action, minimising latencies as far as possible in an IRQ-based system. Moreover, XDP is flexible enough to fit into a wide variety of packet processing stacks.

[18] This can present a bottleneck for the amount of code offloaded to this stage in NICs without support for several receive queues—the XDP hook must meet strict timing constraints. 1 Gbit/s integrated NICs present this problem, in my experience.

This is accompanied by the `AF_XDP` (Corbet, 2018) socket family, which may circumvent the remainder of the network stack by sending the packet directly to user code from an XDP hook. As a notable application, the main packet path of the OVS software switch has been migrated to `AF_XDP` due to its performance and ease of injecting user code into the kernel stack (Tu *et al.*, 2021). The file descriptors of these sockets are placed in an eBPF map, enabling faster packet processing by reducing the time delta between the fast and slow packet paths. User and kernel code share a set of ring buffers, queues, and a large block of preallocated UMEM to pass packet-sized buffers between another in response to completions and demand, a mechanism that is outwardly similar to netmap. While map sizes are fixed at compile time, sockets have a many-to-one relationship with the actual hook program, enabling per-core or per-application service of packets to make better use of

cache coherence. `AF_XDP` sockets support both poll- and IRQ-mode at the user level.

Experimentally demonstrating latency benefits We can illustrate the relative performance of these frameworks against one another (and the traditional `AF_PACKET`) using a simple microbenchmark. We can measure approximately how much time each 'cut' into the kernel saves by connecting two identical machines together over a single link (using NIC hardware with support for these technologies). One machine generates and timestamps traffic, while the other performs a MAC swap function using the intended framework to return packets to the first machine to measure sampled RTTs.

To do so, the two commodity host machines (*Src* and *Swap*) were set up using an Intel Core i7-4790 (4 × 3.6 GHz), 16 GiB RAM (DDR3, 1866 MHz), an Intel X710 40GbE NIC, running Ubuntu 21.10 (5.13.0-30-generic). The hardware lacks support for direct cache access functions like Intel DDIO, and clock scaling was disabled for predictable measurement. The machines were connected over a 40 Gbit/s direct copper cable using a single Rx/Tx queue on *Swap*. Traffic was generated on *Src* using Pktgen-DPDK (Wiles, 2021) to generate 64 B packets at 1 Gbit/s, timing 20 000 packets per framework using the included uniform latency sampler. *Swap* was configured in four ways, using DPDK's *testpmd* application for receipt and forwarding:

DPDK *testpmd* was set to swap MAC addresses and forward packets, using the X710 NIC's `i40e` poll-mode driver.

Native XDP A custom eBPF program set to swap packet MAC addresses and always return `XDP_TX` was manually installed.

AF_XDP *testpmd* was set only to forward packets, using the XDP poll-mode driver with a custom eBPF program to swap MAC addresses before redirecting to the first supplied XDP socket. This driver makes use of polling support for `AF_XDP` sockets.

AF_PACKET *testpmd* was set to swap MAC addresses and forward packets, using the `AF_PACKET` poll-mode driver.

eBPF programs were written in Rust (version 1.59.0) using an in-development version of the redbpf framework. Note that in the `AF_XDP` case the choice to perform MAC swapping in the XDP hook (rather than userland) is deliberate, to measure the time taken to reach user code after service by the offloaded eBPF program.

Figure 2.5 shows a clear performance hierarchy between the offload and bypass frameworks: *DPDK* has a 11 401 ns lower median RTT than offloaded eBPF code, which in turn is 2167 ns lower than `AF_XDP`. Distributions of

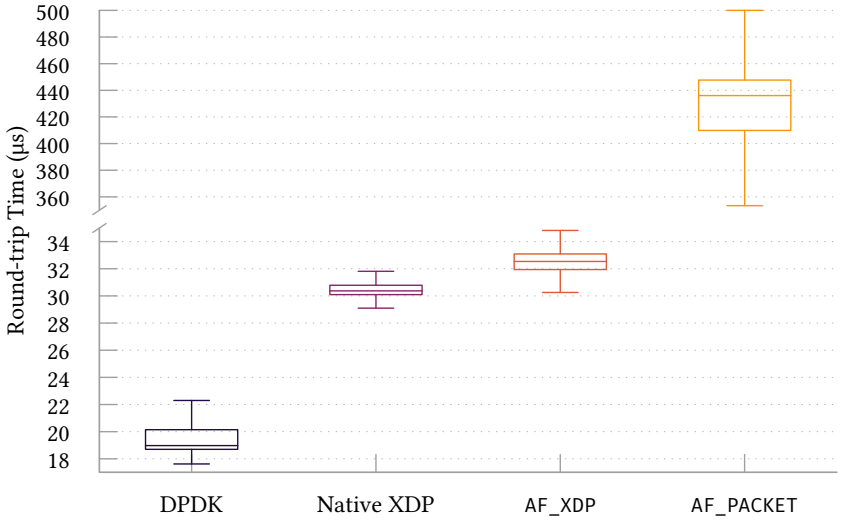


Figure 2.5: A microbenchmark of bypass and offload frameworks' effects on packet RTTs. In line with our expectations from the amount of processing removed by each framework according to fig. 2.4, *DPDK* adds the lowest base latency, followed by *Native XDP* and `AF_XDP`. All of these achieve significantly better forwarding latencies than naïve use of `AF_PACKET`.

latencies fall within a tight bound of 3–5 µs, with *Native XDP*'s RTTs appearing less varied than `AF_XDP`—it's difficult to make any conclusive observations here due to variability in the send and receive stacks of both machines. The inclusion of `AF_PACKET` is, to some extent, a strawman. It does however demonstrate quite succinctly the absolute worst-case behaviour of not making use of kernel or stack bypass technologies for generic packet processing—order-of-magnitude worse latencies and tail behaviour. It's worthwhile to remind ourselves that XDP hook code can run alongside the standard Linux networking stack, benefiting applications without needing to reconsider and rewrite their socket handling code. Additionally, by swapping testpmd for a Rust-based receive stack and reading from the `AF_XDP` socket (using blocking I/O rather than polling), early measurements saw a 39–50 µs gap between eBPF and userland code.

2.3.2 Frameworks for automatic offloading

One of the strengths of the above offload technologies is that their architects are keen to see them achieve widespread adoption, and as such they tend to be well-integrated with existing technical stacks. For instance, eBPF has been offered as a codegen target for both the LLVM and GCC compiler suites since 2015 and 2019 respectively (Edge, 2020; Larabel, 2015), enabling code written in popular high-level languages like C or Rust to be compiled to this level. This presents an interesting research challenge: adapting these developments towards lower-level fragments of such programs intelligently,

such that offloading can improve performance with little input from the programmer. I provide here a brief overview of the higher- and lower-level tooling developed by the research community to either lower the barrier for adoption (and easily migrate existing NF solutions), or to improve on the potential performance benefits.

Host-to-SmartNIC *Floem* (Phothilimthana *et al.*, 2018) presents a *Domain-Specific Language* (DSL) in Python for Click-like dataflow programming to be offloaded, specifying a processing graph of logical blocks which compiles to C code for hosts and target NICs. Users specify which parts of the processing graph execute on each offload device, making this a useful (but not necessarily optimal) tool for investigating offloading strategies. Per-packet metadata is user-specified, but the compiler can infer which state must cross CPU-to-NIC boundaries with the aid of annotations. Individual blocks (and the centralised queue handling) require user implementation in C for each target device class.

iPipe (M. Liu *et al.*, 2019) runs C language programs on SmartNICs and hosts according to whether traffic is at risk of suffering from SmartNIC resource contention. Although iPipe aims to maximise the amount of traffic served by the SmartNIC, processing is dynamically 'unoffloaded' back to the host machine if the SmartNIC's monitored load is too great (i.e., if packet latencies increase). In contrast to Floem, iPipe uses an actor programming model to allow for more dynamic control flow between program blocks; this runtime 'unoffloading' and migration occurs at actor-level granularity. iPipe assumes identical language support between the host and offload target. While C language support is common, identical semantics aren't guaranteed depending on how tailored the target SmartNIC requires code to be. The design is effective enough to protect underlying traffic while achieving improved latency and throughput bounds over a similar DPDK dataplane.

Gallium (K. Zhang *et al.*, 2020) converts C++ Click programs to automatically leverage PDP resources between several segments—pre- and post-host offloaded P4 segments, sandwiching a single host C++ program. This model gives greater flexibility than C-based offloading by enabling program division between, say, a Tofino switch and its attached controller CPU, however this imposes greater restrictions on what logic may be offloaded. Gallium uses LLVM IR to determine read-write dependencies between variables and basic-blocks of the packet processing chain's control flow graph, and account for PDP hardware's capabilities around packet reads (i.e., PDP datapaths typically can't access packet bodies past ~300 B).[19] By annotating these blocks based on their ordering requirements and offload capability, as well as limiting metadata movement to under 100 B, they generate the desired program splits by maximising the IR instructions moved into P4 code—ideally identifying fast paths if there exist cases where the host part can be elided. This approach successfully achieved higher throughput and lower

[19] SmartNICs do not have such limitations, so this is mainly a consideration for PSA switches or more general deployment. Such access is still limited to `externs`, and may need to retrieve packet data from larger, slower blocks of RAM.

latency than a purely host-based FastClick solution for trojan detection. The main drawback is that Gallium requires some annotation to translate Click primitives into MATs as well as read or write dependencies on function parameters, otherwise the conversion to P4 minimises the additional work per target device.

eBPF in the network As a solution to OpenFlow's limited action set, *BPFabric* (Jouët & Pezaros, 2017) proposed that eBPF programs should be used in place of explicit MAT definitions. eBPF was seen as well-suited here because of its simple semantics and restrictions which kept it non-Turing complete; thus, having bounded execution times suitable for real-time packet processing. Its authors keep most of the OpenFlow machinery intact—the device-controller relationship specifically—but instead install one eBPF program per device, encoding its entire view of the dataplane. MAT layouts and program needs are included in the ELF metadata of supplied eBPF programs, and the return value of such a program is the desired output port (including special OpenFlow-style ports to forward packets to the controller). Such programs would be compiled to from a constrained, high-level, C-like language with the typical eBPF restrictions on loops, with a device-local loader playing the role of the Linux kernel's verifier. The proposal contains one particularly forward-thinking aspect—eBPF maps would be mutable from within the target device, enabling and accounting for switch- or NIC-local updates to table state without the aid of a controller. It did not, however, solve the problem of how SmartNICs or other target devices should actually implement the required eBPF execution engines.

hXDP (Brunella *et al.*, 2020) designs a dedicated CPU on FPGA hardware tailored for eBPF program execution on network traffic. Given that hXDP is tailored towards running unported XDP programs, this coprocessor is augmented with dedicated support for helper functions and memory for maps. Their model iteratively runs an expanded eBPF ISA rather than converting programs into a complete pipelined circuit to offer faster redesign and reinstall times: complete circuit planning takes a long time, while JIT compiling an eBPF program into their new ISA is relatively quick. Additionally, their compiler tracks functional dependencies to encode the program in VLIW form to maximise hardware parallelism, while also applying network-specific optimisations. This dedicated design outperforms hosts and NFP SmartNICs in latency, but typically exhibits worse throughput due to its slower single core.

eBPF in hosts Preliminary work has been proposed on automatically splitting eBPF programs between an XDP part and userland part (Shahinfar *et al.*, 2021). This approach considers both horizontal splits, i.e., subdividing code into eBPF chunks using tail-calls, and vertical user↔kernel splits of code. This appears to have promise for increasing application throughput,

although optimal splitting points vary based on the use case.

Morpheus (Miano *et al.*, 2022) offers a more sophisticated form of JIT compilation for eBPF and DPDK programs—using lightweight sketch-based measurement to drive profile-guided optimisation and recompilation on a regular time interval. This produces compiled bytecode tailored to the observed traffic distribution, twinned with inlining of tables into fast and slow paths dependent on MAT contents. While most of this optimisation is provided as-standard when working with profiled LLVM IR, Morpheus provides compiler plugins to account for key features such as MAT logic and match classes. eBPF programs use guard mechanisms to fall back to deoptimised program code as required (MAT or profile changes), while userland code is broken into smaller optimised chunks that are be atomically updated via a trampoline function. Morpheus achieves substantial latency and throughput improvements on larger dataplane programs such as Katran (Facebook Incubator, 2020), in spite of the additional overhead of adaptive trace monitoring.

FPGAs and the wider network *ClickNP* (B. Li *et al.*, 2016) presents an approach for migrating entire Click processing graphs to NetFPGA devices. While tools to convert C programs into the required VHDL specifications exist, the authors find that they lead to suboptimal code in area and *Look-up Table* (LUT) usage. As is standard in Click-like approaches, functions are written as directed graphs of predefined Click-like blocks, in this case each specifically written in a hardware description language to achieve lower FPGA resource utilisation.

Metron (Katsikas *et al.*, 2018) builds on OpenBox to break VNF chains into stateless and stateful logic: stateless processing is offloaded to the network via the ONOS controller (making use of heterogeneous OpenFlow and P4 hardware), while function chains are dynamically allocated one per core. Crucially, the performance of these stateful functions is maximised by using the weak MAT programmability of modern NICs to control core steering dynamically (and consistently) according to load and to balance traffic across replicated functions. Chains are allocated in the network using a combination of topology and current load information (preferring local processing).

Flightplan (Sultana *et al.*, 2021) splits a P4 program into subchunks, placed and routed between heterogeneous PDP hardware along a path—FPGAs, hosts, servers, NPUs, and ASICs—for pipelining (i.e., performance) or redundancy. Flightplan's compiler breaks its input dataplane's IR into blocks according to user annotations in the supplied P4 code. Further user-given annotations denote manual implementations of specific `externs`, to enable device-specific acceleration of key functions such as compression or error correction. Their disaggregation procedure inserts logic before and after splits to handle metadata and state passing. These blocks are statically analysed to extract the data dependencies between sub-functions, which are then composed into a chain over the routing infrastructure.

2.4 In-network compute use cases

To make the value of PDPs and in-network compute clearer, I present here a selection of specific applications that have been improved or enabled outright by these new capabilities. These include advances in network monitoring and telemetry, application acceleration by in-network services, and how the network may allow improved or more dynamic transport and routing. Finally, as it is of particular importance to this thesis and the overall goal of DDN, we shall examine in close detail ML-related use cases. It must be said that the works shown here are indicative, rather then exhaustive; benefiting the system designs in chapters 4–6 by demonstrating the new kinds of network data and state we can use as (on-path) inputs to DDN systems. Interested readers should also examine dedicated surveys such as that of Kfoury *et al.* (2021).

2.4.1 Network monitoring

Traffic monitoring solutions built on non-PDP fabrics can be imprecise or limited. Routers typically provide sampled data such as sFlow, Netflow, or IPFIX, along with imprecise timing at the µs and ms-level (Aitken *et al.*, 2013; Claise, 2004). While this offers useful insight at the aggregate or per-flow level, fine temporal or transient dynamics are lost, such as queue states or per-packet arrival timestamps which might help identify bursty flows or the culprits of microbursts on the network. Precision is not the sole culprit here, µs-level per-packet data simply has too much volume in both raw bytes and packets to meaningfully export beyond the device. PDP fabrics offer here new ways to gather, aggregate, and process per-packet, -device, and -port state in-situ.

Sonata (Gupta *et al.*, 2018) splits dataflow queries on packet streams between PDP hardware and Apache Spark-based collector host machines. These dataflow queries are described in the usual functional style (maps, filters, etc.)—Sonata aims to maximise processing and data reduction performed by the dataplane to ease the burden on stream processing host machines. This includes ILP-based dynamic query refinement to filter out unneeded packets as early as possible. *Snappy* (X. Chen *et al.*, 2018) detects microbursts—transient spikes in queue occupancy leading to packet loss—by maintaining sketches to estimate the queue occupancy of long flows. The input data, packet arrival and departure events, are naturally too numerous to track externally, numbering in the millions or billions of packets per second at switch scale. Long flows dominating output port queues are determined as the root cause of these events, and such heavy-hitter flows may be marked, dropped, re-routed, or rate-limited as required. *Dapper* (Ghasemi *et al.*, 2017) implements in-depth, per-flow analysis of TCP traffic in pure P4. When required, Dapper follows lightweight measurement of per-flow byte counts

and timestamping with more in-depth estimation of congestion and receive window sizes, host reaction times, and latencies to determine whether the sender, network, or receiver is the true bottleneck (without inspecting state on either end-host machine).

A particularly noteworthy family of measurement techniques associated with PDPs is *In-band Network Telemetry* (INT) (The P4.org Applications Working Group, 2020). INT allows for network state—including formerly inaccessible state such as egress queue/buffer occupancies and traversed paths—to be collected and reported without control plane intervention. To do so, packets contain header fields marking 'telemetry instructions' to be executed by participating devices along their path. Typically, requested measurement data is appended to (or written into a dedicated space in) an INT header attached to the packet, piggybacking onto existing traffic, thus keeping packet arrival rates fairly constant. INT sinks then strip this auxiliary data from packets, collect it, and report it to the control plane; enabling finer-grained measurement of network state in routes as well as network-assisted CCAs and load balancing schemes. To some extent, this is a specialisation of *tiny packet programs* (Jeyakumar *et al.*, 2014), which aimed to enable the same end goals and applications by taking a somewhat 'active networking'-like tack—including simple load and push instructions for switch values inside every packet. Instead, INT focusses on collecting pre-defined metadata such as node and port IDs, buffer states, and link utilisations. These fine-grained measurements can be added per-hop, also enabling packet paths to be traced without contacting the control plane as required by *NetSight*'s postcarding (Handigol *et al.*, 2014). At the same time, these collected measurements allow operators to build a full-network picture of per-hop latencies, utilisations, and buffer states. While powerful, INT is expensive to deploy. Although extra per-packet overhead can be bounded for some operations, INT's typical hop-by-hop recording causes a linear growth in packet sizes according to path length. Network MTUs limit the amount of per-packet state which can be added for larger packets. To combat these difficulties, *PINT* (Basat *et al.*, 2020) introduces aggregation and approximation mechanisms to bound per-query values below a certain budget of bits. Most notably, per-hop data collection is uniformly randomly divided over a packet's path, combined with numerical approximations and the use of sketches for per-flow state on switches themselves. *INT-label* (E. Song *et al.*, 2021) aims to reduce redundant measurements and curtail adverse effects on flows by attaching a labelling interval to every port, which can be adaptively altered to account for losses. This includes a probabilistic component such that packets with INT labels are more likely to receive and carry additional metadata (shielding most packets from the costs of INT). *LightGuardian* (Y. Zhao *et al.*, 2021) reduces the impact of exporting per-flow sketches by dividing them into *sketchlets* over several packets, to be reconstructed by INT sink nodes.

PDP hardware also offers a toolkit for the diagnosis and mitigation of more

specific network problems. By using INT-like switch tracking, *Unroller* (Kucera *et al.*, 2020) detects persistent routing loops. Unlike INT, Unroller tracks only the minimum ID seen by a packet, periodically overriding this at geometrically increasing intervals. To detect DDoS attack victims in pure P4, *INDDoS* (Ding *et al.*, 2021) uses count-min sketches to estimate the number of flows inbound for a given server. *Jaqen* (Z. Liu *et al.*, 2021) offers a framework for PDP-augmented ISP networks to detect and mitigate many volumetric, amplification, and semantic DDoS attacks (though not LFAs). To gather state, it maintains sketches of source/destination *Internet Protocol* (IP) addresses and ports alongside counters, as well as useful primitives like native stateless P4 implementations of `SYN` cookie functions and approximate Bloom filter allow- and block-lists. State is polled by the control plane, which interprets sketches to perform higher-level logic (e.g., observing thresholds, significant changes) and installs detection and mitigation functions using a heuristic algorithm to ensure sufficient coverage.

2.4.2 Service acceleration and offloading

Distributed data structures, such as key-value stores, are a foundational part of many data centre applications for process coordination and concurrency control. *NetChain* (Jin *et al.*, 2018) and *FLAIR* (Takruri *et al.*, 2020) offer P4-based implementations of distributed key-value stores, routing and serving requests and updates subject to a chain replication protocol for reliability, achieving an order-of-magnitude latency improvement for reads and writes. NetChain in particular uses a mixture of MATs and registers to accelerate index lookups and store data respectively, though both eliminate steering costs and host stack costs. As another example, distributed hash tables like *No-hop* (Hügerich *et al.*, 2021) see improved lookup times from route caching enabled by the PDP ecosystem.

Other advances built on PDP infrastructure have made it easier for network administrators to scale their infrastructure up to account for additional hosts, handling the associated routing, load balancing, and subscription at lower cost than prior methods. Facebook's open-source *Katran* L4 load balancer (Facebook Incubator, 2020; Shirokov & Dasineni, 2018) makes use of XDP to co-exist on existing service nodes, offer low-disruption updates, and presents lower latency than userland solutions. No performance numbers are shared, though it is reportedly used aggressively in production. *Cheetah* (Barbette *et al.*, 2020) implements consistent load balancing for arbitrary functions at switch scale on Tofino hardware, where it achieves strong tail performance guarantees for flows even during active churn. An intriguing use of PDP hardware that has been presented is how it may allow *packet subscriptions* (Jepsen *et al.*, 2020)—networks may offer an accelerated, Apache Kafka-style publish-subscribe architecture. This includes higher-level tooling to synthesise a complex, content-driven dataplane at runtime (including broadcast, replication and routing).

There have also been key advances in using PDP hardware for stream processing and fusion to accelerate management and monitoring as well as application-level use-cases—including some realisation of the earlier discussion on protocol boosters. *DAIET* (Sapio *et al.*, 2017) proposes performing the *aggregate* step of *partition-aggregate* workloads (i.e., MapReduce) by organising workers and programmable switches into processing trees. Hypolite *et al.* (2020) investigate how to make good use of (NPU-type) SmartNIC parallelism to perform DPI and payload matching tasks through *DeepMatch*. By making good use of the NFP's parallelism and memory models, they offer fast, flow reorder-aware processing by the Aho-Corasick algorithm. *Bolt* (S. Wang *et al.*, 2021) then implements Aho-Corasick on Tofino switches using MATs—however, this requires full recirculations to extract a fixed size byte slice per pass. To reduce traffic impact on infrastructure while freeing host resources, *ZipLine* (Vaucher *et al.*, 2020) uses packet checksum functionality on Tofino switches to implement transparent traffic compression.

2.4.3 Transports, protocols, and routing

Making better use of networks through TE, more sophisticated CCAs, and more adaptive routing is a challenging but worthwhile endeavour. PDP hardware allows us to place more complex decisions and new algorithms at a lower level, where they can take advantage of the fine-grained or complex state and measurements that in-network compute gives us access to (section 2.4.1). Our ideal aims are that flows would converge to their optimal fair-share of bandwidth faster (and with greater accuracy), better respond to congestion and incast events, or more dynamically make use of network capacity. Moreover, it allows us to consider new schemes for doing so as networked use cases change or evolve over the years to come. We'll consider here some ways in which PDP hardware makes the management and operation of networks faster or more resilient by measurement, cooperation, or novel algorithms.

Multipath networks are commonly used to achieve high bisection bandwidth between hosts, but require specialised schemes like *Equal-Cost Multi-Path routing* (ECMP) to balance load across their fabric. PDP hardware is configurable enough to allow the development and deployment of new routing protocols which are more dynamic, fairer, and uniform; ideally, with less control plane interaction. *HULA routing* (Katta *et al.*, 2016) provides a P4-based solution targeting data centre networks, using periodic broadcast probes sent in-band to update every switch's estimate of its best path to every other destination. *Contra* (Hsu *et al.*, 2020) extends this with a custom policy language to encode constraints and more involved routing criteria (i.e., to support arbitrary topologies): probes also obey these rules in the reverse path. Both approaches achieve substantial improvements to fairness above ECMP, and scale well by tracking only a single cost per destination.

Active Queue Management (AQM) allows for flow and packet priorities to be expressed for their egress from a switch, enabling shaping or TE to ensure optimal QoS or *Quality of Experience* (QoE)—for instance, prioritising the packets of short flows (minimising the risk they are dropped by a full buffer) to minimise relative *Flow-Completion Times* (FCTs). *PIFO* (Sivaraman, Subramanian *et al.*, 2016) queues allow for individual packet priorities and dispatch times to be determined by the ingress pipeline of a PDP switch's MATs—offering almost arbitrary programmability—but require this dedicated hardware primitive. *PIEO* (Shrivastav, 2019) queues extend their expressiveness further by allowing arbitrary elements to be dequeued, also. However, the reality is that commodity P4-based switches aren't yet suitable for enabling many of the AQM disciplines developed by the research community. Kunze *et al.* (2021) find that AQM solutions must be co-designed for the target environment, for instance pipeline and register access constraints make it impossible to express some algorithms. Implementation of AQM schemes like *PIE* (Pan *et al.*, 2017) is found to require numerous tradeoffs, with meaningful performance costs in each case.

CCAs play a key role in ensuring that congestion-aware transport protocols are able to make use of their maximal fair share of bandwidth in the network. Original *and* state-of-the-art CCAs in use on the wider Internet, such as TCP BBR (Cardwell *et al.*, 2016), operate on minimal information transfer between hosts or endpoints, and are often reliant on congestion *signals* from the network like packet losses and changes to RTTs. This is a necessity to minimise the network costs of transferring state, to co-exist with heterogeneous CCAs, and to isolate their logic from the network itself to prevent wider ossification. This has significant drawbacks. Advanced, control-theoretic CCAs like the *PCC* family of CCAs (Dong *et al.*, 2015; 2018) or *Copa* (Arun & Balakrishnan, 2018) are computationally expensive compared to their forebears. Having more, high-quality information also prevents fairness issues which may arise from needing to rely on carefully tuned responses to otherwise opaque signals from the network, such as BBRv1's noted unfairness with other TCP traffic (Ware *et al.*, 2019). Even older SDNs show the value of high-quality, global information—consider *OTCP* (Jouët *et al.*, 2016), which calculates optimal pairwise TCP parameters by using OpenFlow and its supporting protocols to actively measure static network characteristics like latencies and link bandwidths. PDP infrastructure can go further still, and offers new ways to offload CCA logic or to augment it with assistance from the network. Data centre networks are best placed for this—*NDP* (Handley *et al.*, 2017) is a receiver-driven CCA where P4 switches truncate packet payloads when buffers are at risk of exhaustion, while senders choose network routes for fairer load balancing. Header-only packets are prioritised, providing a dedicated fast path for control packets and effective congestion signals. *HPCC* (Y. Li, Miao *et al.*, 2019) instead uses the INT techniques discussed above to expose path characteristics and loads to a connection's sender. `ACK` messages carry their packet's INT data; the sender adjusts its target rate per-

ACK (minor) and per-window (major). As this specifically accelerates remote DMA, endpoints require FPGA-enabled NICs. To enable offload of arbitrary CCA logic, Arashloo *et al.* (2020) present *Tonic* as a framework for expressing the *transport logic* (what data segments are sent at what times) while handling connection and buffer management. These may be modified via small user-programmable blocks containing ALUs with access to per-flow state. The remainder of their framework handles (de)packetisation, packet transmission for many waiting flows, and DMA handling.

2.4.4 Machine learning

PDP hardware and in-network compute give us the capabilities to perform packet and flow classification (*inference*) at line rate, and to accelerate data centre-scale *training* of ML models for both DDN and other domains. As we'll examine throughout chapter 3, ML techniques make it possible to learn and act upon complex patterns or relationships in network data to perform all manner of management and optimisation tasks more effectively. While I cover specific implications of data formats and training algorithms later (sections 3.3 and 3.4), the gist is that machine learning inference and training are generally dependent on floating point data (as training gradients are real-valued), peripheral accelerators like GPUs or *Tensor Processing Units* (TPUs), and are computationally complex. Host machines are, then, in many ways the best-suited location for ML training and inference at scale, but these come at the cost of network and PCIe costs to reach the host's CPU, and then further data transfer costs over PCIe to the GPU. Inference at the host thus presents real tradeoffs: either choose throughput by batching queries and new data for the GPU, or choose consistent (tail) latencies by using the CPU. This becomes all the more interesting and challenging when we consider the additional classes and finer granularity of measurement data exposed by PDP hardware, such as those discussed earlier in section 2.4.1. Of course, use case depending, this raw data is produced at volumes and rates far too high to meaningfully move across the network. Host machines can't handle raw packet-per-second demands at this scale, let alone when every packet must undergo some ML algorithm. Per-packet inference (i.e., for security, QoS, or TE classification) is thus untenable in this framework. Ideally we would process this in PDP devices, but this is at odds with their resource-limited nature—limited ALU capabilities, a definite lack of floating-point hardware, and small amounts ($\mathcal{O}$ (KiB–MiB)) of high-speed memory. This leaves us with a difficult dilemma. In-network ML research aims to solve this (preferably at line-rate) by focussing on the modifications needed to install inference techniques onto PDP hardware—tailoring algorithms and data formats to the target device.

Inference *IIsy* (Xiong & Zilberman, 2019) investigates per-packet inference using pre-trained classical ML models on programmable switch hard-

ware, as well as methods for cheaply updating models at runtime through the control plane. In particular they use custom parsers as a feature extraction mechanism, and convert parameters and inference logic into MATs to implement these models using pure P4. This guarantees their compatibility with the vast majority of PDP devices in deployment. These include multiple implementations of SVMs, decision trees, Naïve Bayes, and K-means classifiers, executing but not training on these devices. In particular, they investigate the best use of tables in these scenarios—per-feature, per-class and per-cluster—noting that all cases exhibit different impact from quantisation and processing length (in recirculations).

NNs require more specific considerations due to their variable structure, but their implementation in PDP hardware is an important topic due to their ubiquity in ML and DDN research. Langlet (2019) has shown the viability of NN inference using 64 bit quantisation on NFP SmartNICs, but this can observe high (~500 µs) inference latency on line rate traffic for larger inputs. This compute model has the main downside, however, of being unsuited to RMT or PSA switches. *Binarised Neural Networks* (BNNs) have been shown to be a promising data format for PDP hardware; not least because they are computationally efficient, but because they admit ready conversion to MATs, and are thus suitable for deployment on *all* P4 devices. *BaNaNa SPLIT* shows how these BNNs, installed in NICs, can as a partial offload mechanism (Sanvito *et al.*, 2018); DNN inference is often carried out on the CPU to remove latencies imposed by GPU batching and transfer, but they show that the fully-connected layers of such networks can be accelerated further by NICs subject to some accuracy loss. *N2Net* (Siracusano & Bifulco, 2018) and *N3IC* (Siracusano *et al.*, 2020) examine, collectively, different ways of running BNNs among SmartNICs, P4 implementations, and FPGAs to achieve line-rate, in-path, per-packet inference of classes for use by later tables or packet tagging.

Taurus, examined earlier in section 2.2.4, allows for implementation of the inference mechanisms above closer to line-rate without needing to substantially alter input and policy data formats. At fabrication time, Taurus may be configured to use the required fixed-point depth. Other specialised hardware includes *BrainWave* (Fowers *et al.*, 2018)—an NPU designed solely for parallelised SIMD NN inference—which reduces batching by 32 × compared to GPU acceleration. However, inference still requires $\mathcal{O}$(ms) in representative use cases (Duarte *et al.*, 2019). Neither, however, allows for on-device training; Taurus instead sends sampled data to the controller, while similar provisions could apply to a BrainWave installation.

Training On ML training, most research on how PDP hardware may be applied falls into the optimisation of *distributed ML training* via in-network aggregation. The main use case here involves many nodes in a network exchanging gradient vectors to be combined with a shared model—however,

latency-optimal schemes based on central *Parameter Servers* (PSes) have strongly synchronised inbound traffic which causes high, bursty losses and significant bandwidth consumption near the traffic destination. Aggregating these gradients is simply vector addition, which any interim node may perform to reduce the upstream traffic burden. PDP hardware provides a sound basis for the kinds creative co-design needed to ensure optimal training behaviour here, making better use of the network *and* improving final training times and performance. At a high-level, *iSwitch* (Y. Li, Liu *et al.*, 2019) uses the plasticity of FPGA devices to implement a dedicated floating-point adder and gradient storage; gradient packets are signalled to the dataplane using a reserved pool 2 *Differentiated Services Code Point* (DSCP) (Baker *et al.*, 1998) value, triggering their aggregation (and eventual retransmission to the PS). Solutions targeting general PDP switches—e.g., *ATP* (Lao *et al.*, 2021) and *SwitchML* (Sapio *et al.*, 2021)—instead must rely on fixed-point data formats to enable this in these environments. Unfortunately, to the best of my knowledge true on-device model training for PDP ML or RL hasn't yet been examined beyond the work I present in chapter 5.

2.5 Summary

I have described the history of programmable networks from their initial forays to-date, including what is now a rich tapestry of software-defined routing and bespoke programmable devices mixed with heavily-optimised host machines. We've examined how advances in dataplane programmability enable application performance to be meaningfully improved—lower latencies and higher throughputs—by taking clever advantage of PDP hardware. In particular, this covers the importance of moving high-performance services to the right execution environment—*offloading*. Most crucially, considering in-network compute from the outset enables a plethora of ways to meaningfully improve the management of today's networks.

My concluding thought is that PDP and its use cases present a vibrant, ongoing line of research—particularly when we consider how it can be combined with DDN. It offers today a source of raw device data which we would have never been able to feasibly use or export, as well as the necessary tools to perform complex data analysis at line-rate and switches' scale. At the same time it provides ways of helping host machines scale further via aggregation, and a promising location to perform complex, data-driven logic and reaction. I am of the opinion, however, that it is still a field in its adolescence. The hardware solutions and designs which have arisen and entered widespread market adoption (e.g., RMT→Tofino) are impressive, powerful and capable—but they are the first wave of fully programmable devices at this scale and this form-factor, and we should expect even more innovation in hardware and languages as the field evolves. This may well take the form of incoming heterogeneity, as device manufacturers produce PDP devices

tailored to different use cases in much the same vein as *Taurus*. Projects like *BrainWave* reiterate that network hypergiants already hold the means and motivation to do just this.

Chapter 3

Data-driven Networking

Many aspects of modern networks, such as CCAs, thresholds for differentiating services, and flow classification rely heavily upon hand-tuned heuristics. As in section 2.4, there is still vast scope to improve on communication latency and throughput, or to avoid and work around deleterious traffic patterns (such as incast communication). As a result, research into CCAs, network designs, and routing procedures is very much ongoing. Crucially, as these operations lie at the core of network operation their solutions tend towards extremely efficient heuristic methods; they must be evaluated per-packet or react as quickly as possible to state change. Unlike the approaches empowered by the programmability explored through chapter 2, the norm is that most deployment environments have no access to such tools, and thus have reduced scope for co-design or measuring network properties that would make their design simpler. Designing new methods for network optimisation then requires deep insight into any problem, its edge cases, and the hardware & performance characteristics of the target environment.[1]

[1] This affects how we design such heuristics even on commodity hosts—for instance, kernel-space CCAs (i.e., as part of the TCP stack) are unable to use floating point arithmetic.

Suppose that, as network administrators or protocol designers, we have access to a reasonable amount of information about the machines, network segments or ASes under our control—measurements, observations, and statistics taken at run-time, from simulation, or by modelling. A natural question to ask, then, is whether we can use this data to enhance and improve the operation and use of our network automatically. Thinking further still, we might wonder whether we can outperform the general (yet useful) heuristics which are widely deployed and researched, allowing us to tailor network behaviour according to its environment and traffic patterns. These questions are the founding principles of *Data-Driven Networking* (DDN)[2], a recent field of research focussed on the automatic control and optimisation of network systems, which has sprung forth due to recent advances in ML and RL (Feamster & Rexford, 2018; Kellerer *et al.*, 2019).

[2] Alternatively titled *self-driving networks*.

The ideas and goals of automated network control have always existed and evolved in one form or another, particularly as computational inference and learning have grown more powerful. Primarily, these ideas have propagated

in their early forms via position papers offering a 'vision of things to come'. This was first famously formalised as the *knowledge plane* (Clark *et al.*, 2003), in contrast to the *data* and *control* planes. This proposal captures not only the above concepts of automation as a means for network control, but also for collaborative or commercial sharing of information between end-hosts, transit ASes, and organisations to build up a global picture of the needs of the network. In truth, over the past 19 years we have moved no closer to such a unified substrate, though automated inference based on the data we *do* have is richly researched. A later attempt to combine this with SDN as *Knowledge*-defined networking (Mestres *et al.*, 2016) takes key steps in clarifying the field, through concrete problems and promising ML developments, but effectively curtails the scale of knowledge sharing. DDN itself is named and defined by Jiang *et al.* (2017), who again expand the scope for optimisation beyond network control to include *end-points*; towards application and transport layer optimisation for hosts and servers, as well as control of the underlying fabric.

Starting out with the aim of emphasising and motivating the value of DDN, I discuss and introduce some of the recent developments and applications of ML and RL techniques in computer networking (section 3.1), before then moving onto to explain the 'building blocks' underlying these approaches. Specifically, I introduce relevant function approximations (section 3.2), techniques to learn these representations including RL (section 3.3), and representations for different target devices (section 3.4).[3] Section 3.5 discusses additional challenges inherent to DDN, while section 3.6 then presents an overview of the security perspective surrounding current ML and DDN approaches. Although this context and its challenges are rapidly evolving, an understanding of security issues is key to offering a complete picture of the viability of DDN and the care which must be taken in its research. Sadly, full examination and further development lies beyond the scope of this thesis—it is, in fact, a thesis-worthy topic in its own right (Papernot, 2018).

[3] There are other intersections between networked systems and data-driven techniques which I omit here but deserve a cursory mention: advancements in *Federated Learning* (FL) and distributed ML training, which show how computer networks can aid the wider ML field.

3.1 Use cases

To give a clear (if somewhat informal) introduction to what different processing techniques can offer, I present a selection of DDN use cases. The aims of this section are threefold: to offer a rough intuition of the capabilities of state-of-the-art ML/RL techniques, to present the breadth of optimisation and control problems in DDN, and to describe the sorts of interaction model and co-design required to meet performance guarantees. In the case of RL-based works, I devote extra space towards highlighting the state space ($\mathcal{S}$), action space ($\mathcal{A}$ if discrete, $\mathcal{A}_{\mathbb{R}}$ if continuous), and reward source ($\mathcal{R}$). If live-control approaches are evaluated using network traces instead of a live environment or suitable simulation, I mark them with a '†'—this does not invalidate those authors' findings, but should invite a hint of scepticism

based on the discussions of section 3.5. Mirroring some of the problems introduced through chapter 2, I present a brief critical survey of solutions offered by DDN: network management and optimisation, including classification and TO; transport- and link-level protocol design, primarily of CCAs; security and verification; client- and server-side multimedia optimisation; and resource placement and job scheduling.

The main implication for this thesis is that many network control or optimisation problems can generally be cast as *Markov Decision Processes* (MDPs) (or otherwise fit into the DDN mould) very effectively. However, there are decidedly *right* ways to go about their design, and it is these insights which this section extracts and collates. Readers anxious to see these design elements and takeaways common to this broad spectrum of applications might skip to section 3.1.6. These findings directly feed into design of systems throughout chapters 4–6 concerning how they interact with network data in a scalable way, how problems are formulated to account for inference costs, and how we can use in- or out-of-path compute to its fullest. However, the critical analysis of the *Marl* RL DDoS mitigation strategy (as part of section 3.1.3) is notable for identifying key flaws in its evaluation and design which directly feed into the motivation and formulation of chapter 4.

3.1.1 Network management

Routing and traffic optimisation As discussed earlier, routing is the task of moving packets of network data from their source to their destination, ideally without losing any in transit and as quickly as possible. We can consider this from two perspectives: moving a packet towards the AS where the destination is located using logical boundary information (*inter-AS routing*), and moving packets over the physical infrastructure within an AS (*intra-AS routing*). As inter-AS routing requires consistent protocols and negotiation between organisations, intra-AS routing offers more scope for optimisation and innovation. The usual term for such processes is TO/TE, aiming to minimise congestion and increase client QoS (Elwalid *et al.*, 2002).

Valadarsky *et al.* (2017) show how RL can be used to route traffic by mapping the last k demand matrices ($\mathcal{S}$) into a set of edge weights ($\mathcal{A}_{\mathrm{R}}$). The calculated weights are used to compute probabilistic forwarding strategies based on classical hop-by-hop routing, which then allow predicted congestion to be computed for the following demand matrix ($\mathcal{R}$). This is striking work because it presents an environment where RL categorically beats supervised learning—where predicting a set of actions to take is more effective than predicting the next state and then computing an optimal assignment—and is able to outperform the non-ML *oblivious routing* (Azar *et al.*, 2003) for some problem models. There are several key takeaways from this work: their exploratory designs show that system performance and learning rate rely heavily upon output model size, emphasising the need for a minimal repres-

entation of actions/predictions made;[4] policy execution occurs outside the packet path, and so learns feasibly online; and that using DDN outputs as the input for a well-defined algorithm can offer more interpretability and trust in an optimised system. A drawback worth discussion is their NN architecture's input and output dimensions depend on the network under control ($k \cdot |V|^2 \rightarrow |E|$), and so learnt policies are not portable even under simple alterations like runtime switch and link additions. Memory cost, compute time, and parameter count would equally scale poorly in larger networks.

[4] Even though a smaller model size is arguably less expressive, the fact that there are fewer parameters to learn can be instrumental in converging to a more effective solution more quickly.

AuTO (L. Chen *et al.*, 2018) examines several TO problems in greater depth, explicitly aiming to optimise datacentres of over 10 000 servers via *Deep Reinforcement Learning* (DRL). This presents a key problem: inference using their architecture has a ~100 ms latency, which is rather at odds with the long-tailed distribution of datacentre traffic—namely, that shorter *mice* flows greatly outnumber longer *elephant* flows (Pan *et al.*, 2003). The main consequence is that trying to take per-flow actions in such low-RTT environments causes decisions to either apply late into the flow lifecycle or miss their target entirely, unless they can be reliably taken in less than a millisecond. The posed solution uses two agents concurrently, for mice and elephant flows respectively. *sRLA* produces a set of flow size thresholds for simple queue priority assignment for mice flows[5] ($\mathscr{A}_{\mathbb{R}}$), using the 5-tuple, FCT, and size of each completed flow ($\mathscr{S}$) to optimise the ratio of average per-packet queue times ($\mathscr{R}$). Flows in all but the last priority class are routed using ECMP. *lRLA* then makes bespoke decisions for the remaining elephant flows which—with high probability—will continue long enough to be meaningfully benefited. For all live and completed flows, it uses the 5-tuple with the current priority (if live) or the FCT and size (if complete) ($\mathscr{S}$) to choose the flow's priority, rate, and route as an XPath ID (S. Hu *et al.*, 2016) ($\mathscr{A} \times \mathscr{A}_{\mathbb{R}}$). This is conditioned on the ratio of average throughputs between two timesteps ($\mathscr{R}$). The main design feature of interest to us is this agent separation; that an RL agent can be used to control a time-sensitive system by generating a compact set of parameters for another, more efficient algorithm. However, the reliance on XPath route numbers as an action ties the lRLA policy to the network it was learnt in, preventing shared training in spite of the fixed-size architecture.

[5] Smaller flows are prioritised, as they are assumed to be more deadline sensitive or to suffer higher relative FCTs in the event of losses.

SmartEntry† (J. Zhang *et al.*, 2020) uses an alternate formulation of TE to selectively route traffic at key switches based on its destination. This differs from Valadarsky *et al.* by using the REINFORCE RL algorithm with *Convolutional Neural Networks* (CNNs) to choose a set of location-destination pairs to install route changes ($\mathscr{A}$) from the *current* traffic matrix ($\mathscr{S}$). For these nodes, an ILP model calculates an optimal probabilistic forwarding policy among their neighbours, whose maximum utilisation is used as a loss function ($\mathscr{R}$). Although this outperforms (weighted) ECMP, this has much the same scale and transfer issues as Valadarsky *et al.* ($|V|^2 \rightarrow |V||V-1|$)—in ISP networks this is to some extent acceptable, given that $|V| \leq 49$ in representative trace data. The key concern is that the runtime cost of the ILP

formulation isn't documented, which can have a severe impact on stability if traffic matrices change quickly.[6]

Inter-AS routing in the modern Internet is fairly fixed, operating according to the fixed principles of the BGP suite. However, mapping operator intent into effective, bug-free route announcements presents some scope for optimisation. *DeepBGP* (Bahnasy *et al.*, 2020) uses *Evolution Strategies* (ES) and *Graph Neural Networks* (GNNs) to generate prefix announcements for AS pairs ($\mathscr{A}$) from an input matrix of reachability preferences ($\mathscr{S}$). Each proposed solution is then graded on the number of routing constraints upheld ($\mathscr{R}$)—training continues until a solution is found meeting all constraints. There are, of course, caveats to solving what is fundamentally a *Constraint Satisfaction Problem* (CSP) in this manner. SMT solvers produce outputs faster than DeepBGP takes to train, and it is unclear whether any transferable properties of an input instance are learnt even though raw inference time is faster.[7] As with other CSPs, non-exhaustive solvers are unable to assert whether the input problem is unsatisfiable (and if so, whether the number of constraints that have been met is maximal).

[6] CNN execution is performed *only once* irrespective of how many reroutes are inserted, so this cost is likely dominated by the NP-Hard ILP.

[7] Given that the input and output formats depend on both the high-level intent and the AS relationship graph, the model architecture is intrinsically tied to the given problem.

Flow/packet classification Identifying the type of traffic carried in a flow is a key part of ensuring QoS/QoE guarantees, traffic optimisation, and network security. However, the realities of Internet traffic require that classification is *fast*, contrary to the inference costs typical to DNNs. One stream of packet classification approaches assumes we begin with a full set of enumerated rules ($\sim 10^5$–10^6) and matching priorities, making scalable lookup (i.e., significantly faster than $\mathcal{O}(n)$) a key challenge.

NeuroCuts (Liang *et al.*, 2019) successfully applies DRL to this task. This is, interestingly, quite different from most RL applications in that it *builds a decision-tree classifier* from input rules. To handle the variable size of generated trees, for each non-terminal node the agent uses the min/max bounds of all its inputs ($\mathscr{S}$) to choose both a dimension and cutting/partition point ($\mathscr{A}$). These are fixed-size subproblems, giving a generalised and transferable policy. The set of classifier rules to encode is never passed in as state, only being exposed indirectly via node termination and a tradeoff score between subtree size and depth computed at completion ($\mathscr{R}$). Constructed models have the benefit of being interpretable and fully deterministic. The most clever part of this work is that it keeps the slow DRL work out of the critical path (a necessity for fast, line-rate traffic classification), while learning environment-specific behaviour. DRL is not directly suited to high-rate, low-throughput classification (nor is RL suited to classification versus ML), making this strategy particularly useful. *NeuvoMatch* (Rashelbach *et al.*, 2020) uses several trees composed of small NNs to store lookup information in a more compact way. This effective compression offers improved latency and throughput on x86 hosts as the entirety of each model now fits into cache memory. Rules not captured by these NN trees are looked up us-

ing a decision-tree or other standard packet classifier. This does present a large tradeoff against the above: simpler decision trees can be used natively in TCAMs or admit conversions to MAT structures, meaning that NuevoMatch cannot be trivially ported to network hardware.

In the case that we lack *a priori* knowledge of labelling rules (but do have labelled training data), it becomes straightforward to train and apply ML for classification. Historically, packet bodies have been useful in this task as a variation of DPI investigated using, e.g., *n*-gram models (Yun *et al.*, 2016) and segmented packets (R. Li *et al.*, 2018) as inputs to LSTMs or *Gated Recurrent Units* (GRUs). This is no longer the case in the wild; a key issue nowadays is that encryption of traffic is fairly ubiquitous due to the proliferation of application-level security (HTTPS), secure transports (QUIC) and *Virtual Private Networks* (VPNs)—all of which severely limit the input data we can glean from packets.[8] Using headers alone, there have been successes on common datasets using Naïve Bayes (A. W. Moore & Zuev, 2005), Bayesian NNs (Auld *et al.*, 2007), CNNs (Lotfollahi *et al.*, 2020), and self-attention mechanisms (G. Xie *et al.*, 2020). What is often not masked, however, are application-level timing characteristics of this traffic such as patterns of up/down rates, interarrival times, and statistics gathered over traffic bursts. This additional information makes the task tractable on e.g., *k-Nearest Neighbours* (*k*NN) and decision tree classifiers (Draper-Gil *et al.*, 2016), or LSTMs and CNNs (Aceto *et al.*, 2019). This extends towards passive CCA identification: for window-based algorithms, CNNs have been used to estimate the *cwnd* parameter and observe its reaction to loss events (Hagos *et al.*, 2018), and modern CCAs are handled using both CNNs and LSTMs in DeePCCI (Sander *et al.*, 2019). There are significant issues with these approaches in practice, in spite of their impressive performance. Inference times on one state-of-the-art design (G. Xie *et al.*, 2020) are 180 µs when accelerated using GPU offload, suggesting that throughput and latency guarantees of modern ASes can't be met without aggressive sampling. Some of these input features are also difficult to collect in-network without traffic mirroring and analysis at hosts—which already handle packets at a rate far lower than line-rate network hardware (Gupta *et al.*, 2018). This is particularly relevant for encrypted traffic, as temporal features are often some of its only exposed characteristics.

[8] This ubiquitous encryption also affects non-ML IDSes and anomaly detection use cases discussed earlier, like IDS software like Snort (Roesch, 1999; Snort Team, 2017) or Zeek (Paxson, 1998; The Zeek Project, 2020).

Performance analysis Bayesian optimisation using Gaussian processes has seen some successes in identifying unexpected performance "hotspots" in OVS through *NetBOA* (Zerwas *et al.*, 2019), and cloud instance configuration via *CherryPick* (Alipourfard *et al.*, 2017). This mirrors its successes in ML hyperparameter optimisation (Feurer *et al.*, 2015; Hutter *et al.*, 2011), as this family of techniques is effective at minimising a cost function using limited data (i.e., when there's a high monetary or compute cost to acquire each sample). For optimisation tasks their use is straightforward, but it must be noted that hotspot identification still requires operator knowledge.[9]

[9] In particular, human knowledge is currently needed to show that a so-called adversarial scenario is more than just an expected scaling characteristic; not to mention the subsequent root-cause analysis.

3.1.2 Protocol optimisation and design

Congestion control As introduced earlier in section 2.4.3 (and as a motivating example in chapter 1), the design of effective CCAs very much remains an open topic. The degree of diversity in networks, from long-fat Internet-style networks to dense low-RTT data centres, in buffering and forwarding behaviours of different path segments, *and* the unforeseen interactions between disparate CCA mechanisms, presents a huge problem space to work in. Incorrect assumptions can have knock-on effects in not just overall performance, but in fairness of longer-lived flows to other traffic, or in catastrophic increases to the FCTs of short flows. As a result, automated CCA learning is a particularly attractive prospect; more so when we recall the dominance of congestion-aware traffic in the wider Internet (appendix A).

MVFST-RL (Sivakumar *et al.*, 2019) uses DRL to manage window-based congestion control in QUIC. An agent then controls the congestion window size; incrementing, decrementing, halving, doubling, or keeping its value ($\mathscr{A}$) to optimise throughput and latency ($\mathscr{R}$). In contrast with many prior RL works, their RL agent takes actions asynchronously by coalescing state updates over time, between action choices.[10] Input states are comprised of RTT statistics, byte transmission and receive counts and loss information, combined with the last 5 actions ($\mathscr{S}$). By applying fully-connected NNs followed by an LSTM for policy approximation, this work is competitive with the state-of-the-art due to LSTMs' particular suitability for identifying long-term relations in time-series data. Their work raises again the primary drawback of applying DNNs in latency sensitive applications like CCA design: they observe up to 30 ms action computation time, and have only trained agents via parallel simulation, requiring vast amounts of training data.

[10] The DDoS mitigation use case I develop and describe in chapter 4 uses a similar trick, though this arises due to delayed *reaction* times in the environment rather than inference cost. See section 4.4.6.

DRL-CC (Z. Xu *et al.*, 2019) examines how one RL agent can jointly optimise *Multipath TCP* (MPTCP) subflows and TCP flows. MPTCP differs from traditional transports by allowing data segments in a single logical connection to be sent over several interfaces, who have their own per-subflow congestion control in addition to shared coordination. The state of any (sub)flow is its rate, goodput, RTT statistics, and congestion window size. DRL-CC passes all current states into an LSTM to obtain a fixed-size representation for all flows, which is then combined with the overall state for a target flow ($\mathscr{S}$).[11] Using actor-critic methods, an NN produces a vector of congestion window deltas for all the target flow's subflows ($\mathscr{A}_{\mathbb{R}}$), conditioned on the sum of log-goodputs of live flows ($\mathscr{R}$). Inference latency is kept to a moderate 0.5 ms using the CPU, and performance is comparable to classical MPTCP CCAs on lossy networks—where a high packet loss of 0.5–4 % can be justified by the focus of MPTCP on cellular networks.

[11] This NN architecture, manipulating input state for further action and value networks, is often known as a *two-headed network*. This allows end-to-end training of a feature extraction network and downstream NNs (in this case, the actor and critic networks). Training of the actor and critic component networks jointly improves the base feature extractor.

The *PCC* family of CCAs (Dong *et al.*, 2015; 2018), *Copa* (Arun & Balakrishnan, 2018), and their MPTCP variant *MPCC* (Gilad *et al.*, 2020) offer a control-

theoretic perspective on effective congestion control, improving on heuristic methods. These approaches combine flow throughput, loss, latency and goodput for each (sub)flow into a single utility score, choosing target rates which maximise this score via simple gradient ascent. Although this branch of research doesn't *learn* any function approximation, the fact that operational modes and behaviours are all well-defined allows for convergence to be proven under typical network conditions.

Aurora (Jay *et al.*, 2019) then modifies rate selection in the PCC framework to use a simple NN trained via RL. It computes multiplicative increases or decreases to a flow's send rate ($\mathscr{A}_{\mathbb{R}}$) given an m-long history of latency statistics and loss rates ($\mathcal{S}$). The agent then acts to maximise packet-per-second rate, penalising latency and packet loss ($\mathscr{R}$). By keeping the explicit operational modes of the PCC family, the policies it learns from offline training effectively generalise to unseen network characteristics and designs. However, this formulation was later shown to be unfair to other CCAs (Abbasloo *et al.*, 2020).

Orca (Abbasloo *et al.*, 2020) eschews the "clean-slate" approach common thus far, using a classical CCA (TCP Cubic) as its basis. This decision is empirically and strongly motivated; doing so greatly simplifies the learning task for an RL agent (improving the learnt policy) *and* reduces CPU and GPU utilisation in deployment.[12] Orca tracks m-long histories of a flow's current (and best) throughput and RTT information alongside its loss rate and congestion window ($\mathcal{S}$). Using an actor-critic algorithm, Orca chooses some $\alpha \in (-2, 2)$ every 20 ms, multiplying the congestion window by 2^{α} ($\mathscr{A}_{\mathbb{R}}$), and allows the baseline classical CCA to otherwise act as normal. Each flow acts to improve the current ratio between its current *power* and the best estimate of the Gail-Kleinrock optimal operating point (Gail & Kleinrock, 1981; Kleinrock, 1978)—with some tradeoffs to minimise loss and allow small RTT variance ($\mathscr{R}$). While this naturally requires higher resource use than a heuristic method such as Cubic or BBRv2, this strategy reduces resource costs beyond even the control-theoretic PCC family of CCAs (with better, fairer operation). Reducing the length of time between DRL actions predictably increases resource demands, but leads to better flow performance, allowing a runtime trade-off to be made.

[12] Recall that CCAs almost always control how data is *sent* across the network, and that clients typically send small requests for servers to transmit larger content. This leaves the burden of performing expensive per-packet and per-flow operations with the server, which by this same assumption has to handle many such flows!

Media access control An exciting, perhaps unexpected, network environment is within CPUs themselves—a *network on a chip*—for coordination in multi-threaded programs and ensuring cache coherency in many-core architectures. This design class is necessitated by the limitations of a shared bus at high core counts. Core-to-core communication is either packet-switched using local routers (incurring latency costs) or wireless (potentially leading to collisions). *NeuMAC* (Jog *et al.*, 2021) approaches optimal wireless transmission via DRL. Monte Carlo RL training of a small NN occurs offline from simulation, using complete execution traces. In deployment, an agent

is quantised to 8 bit fixed-precision values on low-latency *Static Random-Access Memory* (SRAM).[13] Each core has a dedicated transmission timeslot, while the agent chooses a list of per-core probabilities every 10 µs to allow transmissions outside this window ($\mathscr{A}_{\mathbb{R}}$), which are halved on a collision. An agent passively listens to broadcast signals, observing the successful transmissions per core and the total number of collisions observed ($\mathscr{S}$), minimising the cycles spent running a program to completion ($\mathscr{R}$). Interestingly, this shows that small, pre-trained, quantised NNs can be placed into core hardware control loops at low latency (512 ns) and low power draw with bespoke integration of NNs into hardware.

[13] See section 3.4 for an in-depth discussion around the topic of embedded ML design decisions such as this.

3.1.3 Security, defence, and verification

Network and computer defence ML and other statistical approaches would seem like a natural fit for the problem of network defence, and have been long-awaited in hope that they might aid automated anomaly detection and the derivation of attack signatures (Bhuyan *et al.*, 2014). Barring some recent exceptions, DDN approaches have languished. In 2010, Sommer and Paxson identified the 'failure to launch' of ML-based anomaly detection systems—a distinct lack of real-world system deployments (Sommer & Paxson, 2010). To quite a large extent, this still holds true today. They posited that their use is made difficult due to significant operational differences from standard ML tasks, including: the high cost of errors and extraordinarily low tolerance for false positives inherent to network intrusion detection (Axelsson, 1999); a general lack of recent, openly available (and high-quality) training data; and diversity of network traffic across varying timescales combined with significant burstiness (Leland *et al.*, 1995). Above the aggregate level, the constant deployment of new services and protocols means that traffic is *non-stationary* and displays an evolving notion of normality (section 4.1.2). Learning is made harder still by the challenges encountered with unlabelled (often partial) data. Moreover, known-poor datasets such as the problematic 'DARPA99' dataset (MIT Lincoln Labs, 2018; Sommer & Paxson, 2010; Tavallaee *et al.*, 2009) and its derivatives such as KDDCup99 or NSL-KDD have yet to be excised from works appearing even today.

Marl (Malialis & Kudenko, 2013; 2015) examines the automated detection and mitigation of DDoS attacks using the Sarsa RL algorithm. As a multi-agent system, Marl agents are distributed at the edges of a network and adaptively learn a policy to control traffic *without* explicit communication or sharing of policy updates. Agents reside at the AS's ingress points, and choose a packet drop probability for *all inbound flows* from the discrete choices $a \in \{0.0, 0.1, ..., 0.9\}$ ($\mathscr{A}$) according to load measurements along their route to a protected server ($\mathscr{S}$). They create a tree overlay topology, subdivided into teams which each receive a separate reward measurement. This aids credit assignment by not punishing teams who contribute little to the total incoming bandwidth. Agents are punished when the network is overloaded

and their team contributes more than its fair share of traffic, otherwise they receive the proportion of legitimate traffic observed at the team leader ($\mathscr{R}$). Applying filtering actions indiscriminately to all flows carried by a switch means that legitimate traffic is easily caught in the crossfire, indirectly harming harmless flows. Although their results appear competitive, their simulation environment uses only congestion-unaware UDP traffic, counter to the realities of Internet traffic as discussed in section 4.1.2 and appendix A. Congestion-aware protocols dominate in many networks; incorrectly applying a packet drop action imposes greater *pushback* (Mahajan *et al.*, 2002) on these legitimate flows than it would on attack traffic. For congestion-aware traffic, this is non-negligible; when packet loss occurs with probability $p \neq 0$, the Mathis equation (Mathis *et al.*, 1997) states that TCP bandwidth is proportional to $1/\sqrt{p}$, while modern TCP Cubic is proportional to $1/p^{0.75}$ (Rhee *et al.*, 2018). Congestion-unaware, *Constant Bitrate* (CBR) traffic then occupies bandwidth proportional to $1 - p$, and in section 4.1 we will show from the literature that volumetric DDoS attack traffic mainly falls into this category. Furthermore, the static overlay topology does not account for the defence of load-balanced or multipath networks, and the reward function relies on either *a priori* knowledge of traffic or an accurate heuristic. These weaknesses are shown more concretely throughout section section 4.4.5, and motivate the design of the *Instant* and *Guarded* RL agents throughout chapter 4.

Other ML techniques have been applied to DDoS detection in the context of SDNs. Braga *et al.* (2010) have shown that *self-organising maps* (an unsupervised, NN-based approach) can act as effective classifiers from flow statistics given ample captures of both normal and attack behaviour. *Athena* (Lee *et al.*, 2017) improves on this through a more generalised (albeit heavyweight) SDN framework for intrusion detection, showing the use of *k-means clustering* to detect individual attack flows. However, their comparison against modern algorithmic DDoS defence techniques such as *SPIFFY* (Kang *et al.*, 2016b) lacks any quantitative evidence.

Most modern malware makes use of evasion techniques or alters its behaviour to appear more benign in the presence of dynamic analysis, such that understanding it (particularly when self-modifying code is used) becomes more difficult for security analysts.[14] *TAMALES* makes use of this principle to great effect (Copty *et al.*, 2018); where most analysis tools aim to mimic a real OS as closely as possible, their "extreme abstraction" relies upon deviating from specifications and expected behaviour to induce anomalous behaviours. Using *random forest classifiers* (Breiman, 2001), they combine static program features with dynamic behaviours observed from buggy OS emulation. The most interesting (and general) feature of this design is that more expensive features and analyses are added to the classifier over time while the output classification remains ambiguous.

[14] This problem long predates the class of evasion attacks on ML models mostly considered throughout section 3.6.1. Evasive malware relies more on introducing *semantic* or *behavioural* differences rather than abuse of decision boundaries in high-dimensional spaces.

Qin *et al.* (2020) attempt to combine the distributed training offered by FL

with the recent advances in BNN use in the dataplane (section 3.4) for attack traffic detection. P4-capable edge switches or NICs host a BNN computed from a local (full-precision) model trained on a co-hosted machine, which communicates model updates to and from a central parameter server as is common in FL. Their work supports the hypotheses that BNNs achieve sufficient accuracy on existing IDS datasets and that overall model convergence makes FL suitable for this type of data. However, this work neither mentions nor considers the central limitation of FL; that edge models need some local means of generating labels for new data.[15] As a result, it's not clear whether FL is even a suitable choice for this task, and so this remains far from a feasible system. However, it is only tested on BMV2 (thus it's unclear how suitable this is for a line-rate system) and relies on in-band table alterations which are only possible in future architectures like the *Portable NIC Architecture* (section 2.2.4).

[15] FL can still be used to train *unsupervised methods* without a local oracle, but clustering and forecasting have limited application for this class of flow filtering.

Verification *P4RL* (Shukla *et al.*, 2019) applies DRL to the guided fuzzing of complete P4 dataplanes. Fuzzing (as opposed to static analysis) allows for the detection of bugs that lie outside of the P4 language itself, e.g., through interactions with the control plane or in hardware-specific behaviour. The key drawback of fuzzing without some manner of guidance, however, is the colossal size of the input value state space. Beginning with a set of invariants extracted from their *p4q* DSL, P4RL iteratively modifies the header bytes of an output packet, starting from an initially valid state ($\mathcal{S}$). The RL agent then chooses a field and value pair (choosing either random values or boundary values known from *p4q*) ($\mathcal{A}$), conditioned on whether that packet violated any given invariant ($\mathcal{R}$). This notably reduces the number of packets needed to trigger any bugs versus a random baseline, but it is not shown whether this reduces the wall-clock time needed to output such a packet.

DeepMPLS (Geyer & Schmid, 2019) applies GNNs towards network verification in the face of link failures for MPLS routed networks. MPLS is commonly used in ISP networks (Vanaubel *et al.*, 2015), and has comprehensive (though slow) tools for discovering routing violations given complex predicates (Jensen *et al.*, 2018). Given that GNNs can be applied to variable-size input graphs, this allows for a useful model to be trained over many instances. This offers two orders-of-magnitude speedup over conventional solvers, with the main caveat that outputs are only 80–90 % likely to be valid, while an algorithmic solution is correct by construction.

3.1.4 Multimedia

ABR video selection Streaming video is a common use case in the modern Internet. Here, users typically want to receive the highest quality video they can, while minimising any noticeable quality changes and the amount of time spent *rebuffering*, which are their core QoE metrics. Servers allow

clients to control this via *Adaptive Bitrate* (ABR) selection, splitting videos into many fixed-length chunks (of 4–10 s) served via *HTTP Live Streaming* (HLS) (Pantos & May, 2017) or *Dynamic Adaptive Streaming over HTTP* (DASH) (The Moving Picture Experts Group, 2021). However, chunk selection is delegated to the client using heuristic approaches such as MPC (Yin *et al.*, 2015). An exciting question to consider is whether data-driven metrics can do better still.

Pensieve (Mao *et al.*, 2017) applies DRL to client-side observations of network state and video performance metrics for effective optimisation of bitrate selection in multimedia streaming. Throughputs and download times for the last k chunks are combined with current chunk and buffer length statistics ($\mathcal{S}$) to choose the next chunk's quality from the standard list (masking any illegal choices) ($\mathcal{A}$). Pensieve acts to maximise an aggregate QoE score, maximising quality[16], while penalising bitrate changes and the time spent rebuffering ($\mathcal{R}$). To reconcile the costs of DRL inference with the limited resources of mobile devices, Pensieve is server-hosted and periodically queried by clients (though it remains effective even under ~100 ms RTT).

[16] This can be a linear or log-linear function of actual bitrate, or can explicitly penalise non-HD choices.

Stick (T. Huang *et al.*, 2020) trains smaller RL CNNs to provide the target buffer occupancy allocated for heuristic, buffer-based chunk selection methods ($\mathcal{A}_{\mathrm{R}}$). This reduces runtime execution costs and retains the interpretable behaviour of traditional ABR strategies. Stick uses the same input as Pensieve, adding in the current reward ($\mathcal{S}$), optimising the (linear) QoE score discussed above ($\mathcal{R}$). To further reduce inference costs, a very small CNN is used to estimate whether the current state is likely to cause a large change in the buffer's target occupancy. Overall, this leads to slightly better client QoE and offers far lower execution costs, completely removing the impact of server RTT as all rate selection can be managed on-device.

PERM (Guan *et al.*, 2020) considers this problem over MPTCP connections, via DRL on standard feed-forward NNs. Modifying Pensieve's state to use per-*port* throughputs over k timesteps ($\mathcal{S}$), PERM chooses both the next chunk's bitrate and traffic splitting proportions over registered links ($\mathcal{A} \times \mathcal{A}_{\mathrm{R}}$). By optimising the linear QoE score with added per-link cost penalities ($\mathcal{R}$), they reduce the use of high-cost links (e.g., 4G). However, their evaluation is unclear in how *only* the underlying video QoE is affected when link costs are disregarded.

Server- and network-driven QoE enhancements *LiveNAS* (J. Kim *et al.*, 2020) extends recent work on offline ML-driven video upscaling towards live content.[17] The main value in doing so is that ML can be applied to both increase user QoE and reduce upstream bandwidth requirements in livestreaming (i.e., in the event that a popular streamer is limited by their own network). This requires a server-side DNN to be trained for each user from high-quality data they are typically unable to upload. LiveNAS solves this training problem by having each sender also encode their stream at a higher quality level

[17] Optimal upscaling relies on having a content-specific DNN, as generic models can still generate noticeable errors compared to the input stream. Successful online training of such a model from incomplete data is an exciting challenge.

than their connection can support. Small high-quality patches with maximal error versus a bilinear upscale are included alongside the lower-quality stream, acting as valuable ground truth for the model to learn from at moderate bandwidth cost. This offers strong QoE improvements, low deviation from the true input video, and can in principle cut the bandwidth requirements for high-profile streamers.

Mangla *et al.* (2020) use several ML techniques to investigate whether ISP or other transit networks are able to estimate video session QoE using cheaper input state such as *Transport Layer Security* (TLS) session lifetimes and flow-level measurements. For instance, the detection of *low* QoE scores would allow cellular networks to provision greater bandwidth or prioritisation to multimedia flows which require it. These methods are most effective at splitting low- and high-QoE flows (with a high degree of confusion in middling flows), suggesting that they could be used as a stage-1 metric to enable more expensive per-packet analysis.

Alohamora (Kansal *et al.*, 2021) uses DRL to generate HTTP/2 asset push and preload policies to reduce page load times over limited networks or on constrained devices. The approach trains offline using LSTMs by grouping page families into clusters, and inferring policies at runtime as needed. Link capacity and RTT statistics, client CPU capacity, and the target page's resource dependency graph ($\mathcal{S}$) are used to output a sequence of push/preload item and prerequisite pairs until an end-token or illegal state is output ($\mathcal{A}$). Alohamora optimises the relative QoE change according to cheap and accurate simulations, with explicit bonuses added whenever a better incumbent policy is produced ($\mathcal{R}$). While considerably more effective than past works on policy generation, the ablation studies shown by the authors indicate a strong dependency on device-specific state (especially the CPU speed of each client). Inference is cheap compared to page load times, around 11–40 ms, offering strong QoE improvements when all input data are known.

3.1.5 Resource placement and management

Job scheduling *DeepRM* (Mao *et al.*, 2016) is one of the first works on simple DRL-based job scheduling among resource-constrained CPUs, aiming to minimise the average job slowdown. What is particularly notable about this work is that it employs intelligent sampling and monitoring while taking multiple actions per timestep. In particular, it maps pixel images of current resource use and the costs of the next k jobs ($\mathcal{S}$) into a discrete set of job choices to schedule and a null action ($\mathcal{A}$). The timestep is advanced on either an illegal or null action, giving an agent a negative reward for all incomplete (arrived) tasks ($\mathcal{R}$).

In reality, scheduling of (data-parallel) jobs is far harder; these are often expressed as a *Directed Acyclic Graph* (DAG) of subtasks with interconnected data dependencies. *Decima* (Mao *et al.*, 2019) applies GNN-based DRL

to completely control job scheduling as part of Apache Spark. To minimise the average *Job Completion Time* (JCT) ($\mathscr{R}$), Decima chooses the next job stage to schedule and the number of workers to be spawned ($\mathscr{A}$) in response to any scheduler events, until all jobs are assigned or executors are busy. Agents use the output embeddings of nested GNNs, processing per-task and executor statistics into job- and system-level summaries ($\mathscr{S}$). To make training feasible[18], episodes are modified to end early in initial training phases. Equally, in any scenario job arrival times are perturbed (maintaining arrival *order*) to prevent excessive punishment due to bursty arrivals. By using smaller NNs at each stage, each decision can be made in around 15 ms.

[18] The Monte Carlo REINFORCE algorithm needs complete execution traces to update an RL policy, but excessively poor policies could take far longer than realistic runs to terminate.

Distributed ML model training is a variation on this problem, typically having high bandwidth costs and JCTs which exist in a trade-off with final model accuracy. *MLFS* (H. Wang *et al.*, 2020) cleverly operates by starting with a heuristic approach to gather samples for DRL training—initially prioritising jobs with faster JCTs or expected accuracy improvements (i.e., fresh training jobs). The agent chooses a set of task-to-executor pairs ($\mathscr{A}$) using the full set of task resource demands and parameters alongside executor utilisation ($\mathscr{S}$). Agents act to minimise average JCTs and bandwidth, and maximise average accuracy, accuracy goals met, and the number of jobs completed before their deadline ($\mathscr{R}$). The RL model is used in place of the heuristic after its policy successfully converges, and simple statistical methods are used to terminate jobs whenever overfitting appears to begin. However, MLFS's execution costs aren't specified, and it remains unclear how it handles (what appear to be) variable-size inputs and outputs.

Some works choose to focus on the simpler (though important) task of parameter optimisation for existing schedulers. The degree of parallelism offers one such 'knob' to tweak in data-parallel job allocation, however it is not one which universally leads to performance gains when increased. In partition-aggregate workloads, coordination overheads dominate if a task is divided between too many workers—*ReLoca* (Z. Hu *et al.*, 2020) successfully trains DNNs to predict job completion from a given worker count and DAG statistics, using a novel sampling method to concentrate training around optimal choices. In the case of independent jobs (e.g., replicated services), *Autopilot* (Rzadca *et al.*, 2020) optimises vertical scaling (the CPU and RAM limits allocated for a task) and the number of workers to minimise user spend versus heuristics. In the former case, an ensemble of simple optimisers (differing by cost model) is used to provide an interpretable suggestion, in the latter case a user-specified strategy is applied to minimise resource use.

Cache management Caching of Internet resources (e.g., webpages or video) is commonly employed by CDNs to serve content to users in a way which minimises latency as well as offering load-balancing for content providers. *RL-Cache* (Kirilin *et al.*, 2020) offers a cache admission policy learnt through

DRL such that the hit rate is maximised. When a resource is requested, RL-Cache chooses to admit or remove that item from the cache ($\mathscr{A}$) based on that object's size, recency and frequency statistics ($\mathcal{S}$).[19] The reward is simply 1 or 0 (hit or miss) per-object in the next batch, divided amongst the previous k decisions ($\mathscr{R}$). Although it is effective after the authors reduce the runtime cost by performing inference only on cache misses, batching is required to meet any reasonable level of throughput. This comes at a cost of latency; a totalled 65 ms for a batch of size 1024, which is comparable to client-server RTTs in the best case (and with tail latencies left unspecified).

[19] Somewhat curiously, the authors choose to quantise these measures into fixed bins; effectively using a one-hot encoding of bin hits for each statistic as the input.

MacoCache (F. Wang *et al.*, 2020) examines a more targeted form of resource caching at cellular base stations via multi-agent DRL to minimise latency and bandwidth demands in mobile edge networks ($\mathscr{R}$). Agents estimate a cache probability for every video file, choosing the top k entries ($\mathscr{A}_{\mathbb{R}}$) based on per-item demand rates and cache status ($\mathcal{S}$). Agents don't share information directly, but do receive a portion of their neighbours' rewards and use neighbours' policies and cache state as further inputs to account for their impact on the overall system.

System and network planning In network planning, ILPs are often used for short- and long-term bring-up of fibre placements and IP route provisioning. *NeuroPlan* (Zhu *et al.*, 2021) uses DRL to suggest a better starting point and prune the ILP search space for this problem, greatly reducing runtimes (typically 3–4 d) in hypergiant networks. The variable size and structure of networks make graph convolution a natural fit, learning to optimise the normalised cost of any new bandwidth provisioned ($\mathscr{R}$) through actor-critic methods. Given the line graph transformation (Harary & Norman, 1960) of the network with IP route capacities as edge labels ($\mathcal{S}$), the agent chooses both a link and the number of discrete capacity units to add ($\mathscr{A}$).[20] Training is accelerated by stopping each episode once either the constraints are met or too many steps have elapsed, and the final link weights are used as new maxima for each ILP variable—this state-space pruning accelerates the ILP by 7–14 × while achieving similar total costs.

[20] Agents are prohibited from removing capacity, aiding learning by making it impossible to regress into an illegal state.

Chip floorplanning, the process of placing and interconnecting FUs for fabrication, can equally be considered as a resource placement, routing, and networking problem. Mirhoseini *et al.* (2021) show that DRL can successfully learn to output compliant designs in around 6 h of datacentre training, compared to months of human iteration. Using Edge-GNNs to process hypergraphs of macros, standard cells, and their interconnections, an RL agent places macros onto a masked 128 × 128 grid ($\mathscr{A}$), from large to small. The feature extraction part of the network processes the complete hypergraph, the dimensions of each macro node, required connections, and associated metadata as inputs for the policy network and value network ($\mathcal{S}$). An agent receives a single reward at the end of an episode: a negative weighted sum of wirelength, congestion and density ($\mathscr{R}$). Effective and fast, this technique

has been put into action in the design of forthcoming tensor accelerator hardware.A more generally useful insight of their work is that completed placements are easy to estimate a reward for; as such, learning the feature extraction network can be bootstrapped as a supervised, offline problem.

3.1.6 Takeaways for effective data-driven networking

Although we've covered a vast, varied collection of problem domains, this selection shows how the tools developed by the ML community can be of great use in the design, optimisation, and control of modern networks. A shared insight is that many of these tasks can be effectively represented (directly, or otherwise) as MDPs, making RL techniques an excellent addition to a network operator's toolkit. Similarly for ML methods, many tasks can be reduced to classification or regression to optimise over a set of parameters. Most importantly for our domain however, these tasks present a broad set of hard and soft deadline demands. Each influences not only where our control logic can run in the network, but the forms of function approximation which are suitable. This is paramount as we move closer to per-packet or per-flow handling, directly shaping any agent's interaction model or control loop.

Effective MDP designs rely on a mixture of in-depth problem knowledge and an acquired general intuition. While this is not something that can be easily condensed, the above use cases have given us common insights into:

- how to design for and around *deadline sensitivity*,
- accounting for a *choice of function approximation* relative to these constraints (and where we wish to deploy and train agents),
- the *scaling and transferability* that are introduced by fixed-size representations (or techniques such as GNNs),
- accelerating training and ensuring higher-level system reliability.

Deadline sensitivity A consistent feature of almost all online works presented here is that complex function approximators, particularly DNNs, have inference costs on the order of 1–50 ms based on model complexity.[21] This has knock-on implications not only on what processing can be performed, but also on what parts of the problem space engineers are likely to consider—observe that many of the above use cases either respond to moderately infrequent events, optimise client-side behaviour, or perform high-level design tasks outside of the day-to-day operation of the network. To see why this is the case, consider that servers and data centre infrastructure must forward and handle thousands of flows per second, and millions of packets per second. As a result, synchronous in-pipeline packet inference increases the

[21] In many cases, it should be stated that batching does increase the throughput beyond the reciprocal of these times, at the cost of high latency and still higher tail latency.

risk of drops or stalled packet transmissions, or delayed response to changes in flow characteristics (possibly followed by reduced QoS or QoE).

We can make the relative impact of these costs more concrete through an example. Suppose that we wished to move inference further down the stack, either to reduce processing latencies, or to work with fine-grained state which is simply too numerous to export elsewhere. As Ethernet moves beyond 40 Gbit/s and 100 Gbit/s, packet processing deadlines grow tighter in tandem; an input stream of 64 B packets demands that a packet be output every 12.8 ns to maintain 40 Gbit/s line-rate. On Netronome SmartNICs for instance, which have 312 P4 pipelines (each as an NPU context), this gives a worst-case 3.99 µs processing deadline for each packet. Of course, this is simpler in reality as real-world traffic patterns tend to comprise a mixture of packet sizes larger than this. And yet, architectures having fewer pipelines are more unforgiving in this sense—the P4→NetFPGA framework (Ibanez, Brebner *et al.*, 2019) has just a single processing pipeline, so timing violations have a higher impact than on an NPU.[22]

[22] This does not presuppose that all processing happens in a single stage, just that the timing of each stage meets this constraint. Recall that pipelined microprocessor architectures allow a designer to meet throughput demands in exchange for latency.

The most clever strategies we've considered work around this limitation by using inference or learning methods to produce the structures, models, or parameters for a more efficient algorithm or heuristic to use. For instance, generating TCAM-optimised matching structures, or outputting only a list of boundary points for degrading a flow's priority by simple comparison—both enabling direct installation to PDP hardware. This agent class still allows for adaptive or environment-specific training beyond what human optimisation can offer, while keeping such execution costs out of the packet path. Ensuring that decisions are taken with reduced frequency and out of the critical data path is the simplest way to minimise the negative impact of inference costs. In contrast, per-packet or per-flow inference is almost entirely restricted to end-hosts, who have lower flow counts than servers and reduced traffic for consideration (~10–300 Mbit/s in typical home networks).

Alternatively, we can make tailored decisions more tractable by narrowing the set of items a system examines in the first instance. Cheap initial analyses (or even simple heuristics) can be used to filter down the set of inputs, identifying the flows, users, or subsystems needing fine-grained optimisation (such as flows which are likely to persist beyond DNN inference latency). On a similar note, this principle can be extended towards input data itself; data-driven methods can be used to go beyond cheaper statistics and trigger more expensive analyses in response to ambiguity at decision time.

Choice of function approximation There have been great advances in accelerating NNs, particularly through the forms of quantisation discussed in section 3.4. In fact, BNNs allow not only for these models to run in programmable switches and NICs, but for per-packet inference to meet the timing

constraints described above. Yet these environments still lack the compute capability to perform backpropagation under these constraints (to update the model in-situ), and to store batches of execution traces (to learn in a stable way). As a result, at present we can deploy a trained (DNN-based) RL agent directly in PDP hardware—but we can only train for this environment using simulations or offline training data. As we shall see later, an answer to this limitation is that we rethink our approach, and use an altogether simpler model or means of function approximation (chapter 5).

Scalability and transferability Most function approximators cannot trivially handle variable-size state, often requiring that such problems are broken down into fixed-size chunks. In the RL case, we've seen that this often involves selecting actions for subtasks within the same timestep—which, of course, introduces further design issues such as which ordering to use to act on subtasks and whether to split reward allocation between concurrent decisions. Tailoring an agent's architecture to the structure of an individual problem instance can sidestep these issues, and is arguably quite suitable in cases which are CSP-like (i.e., iteratively acting to solve a single NP-Hard task). This simplification comes at a cost; requiring full episodic training to arrive at a solution each time, and preventing generalised training between problem instances.

There are substantial benefits to keeping a model's overall architecture (including input and output dimensions) at a fixed size which justify the additional iteration and design work. Using a consistent architecture allows for a model to be trained over many instances and shared between deployments. This is key in not only substantially reducing inference time (as training is no longer required per instance), but also typically improves model accuracy (by learning general features of the underlying problem itself) and allows distributed training in both the federated and parallel sense.

Recent approaches such as GNNs and LSTMs can offer a solution for variable-size problem instances, depending once again on the task of interest. Both allow for the handling of variable length collections of fixed-size elements. GNNs excel at capturing and modelling the *relationships* between objects. LSTMs, however, are suited to *sequences* with an explicit temporal component (i.e., they are sensitive to the input order of samples), making them more effective for mitigating *partial observability* or a non-Markov problem using state histories as an input.

Training and reliability Using a conventional heuristic or algorithm to augment DDN can benefit an agent in several ways:

- Relying on a heuristic method for steady-state operation whilst taking infrequent RL or ML actions can greatly cut runtime CPU and GPU costs, while also providing more reliable or interpretable behaviour.

This is particularly true when actions are new choices for the control parameters of such algorithms.

- If used as a complete replacement during the initial stages of training, then heuristics provide high-quality (and representative) execution traces compared to a random initial policy.
- A heuristic replacement in the early stages of training exactly matches 'normal' behaviour until the policy stabilises, making such systems more suitable for from-scratch training in real networks.

Commonly, many of the approaches we've examined need to deviate from the typical RL formulation to make learning feasible in the face of more difficult environmental behaviour. Consider how many of the above domains must act asynchronously, replay existing traces under slightly varied conditions, take multiple logical actions per physical timestep, or combine and coalesce inbound state whilst policy computations are ongoing. While these changes are rarely justified in the analytical sense, in practice they tend to hold empirical benefit (bearing the caveat that such task-specific modifications may impede learning in another task).

DNNs, particularly as used by actor-critic DRL algorithms, often employ two-headed networks—sharing a feature extraction component between the policy and value networks. Training of these shared layers can be quite effectively bootstrapped offline if, for instance, it is easy to both generate representative state vectors and estimate the reward value (or some other metric) that can be derived.

Although this final point is obvious in many senses, as in many other disciplines we should prefer the simplest, most parsimonious representation which solves a target problem. Even though larger models (i.e., in parameter count) are theoretically more expressive, this impacts runtime costs (in RAM/CPU) and can significantly increase the duration of training needed to achieve convergence.

3.2 Function approximation

The main goal of ML methods is to learn the function between a set of inputs and a set of outputs, for instance mapping histories of flow statistics into classifications in DDN. In most cases, we have some body of *training data*—input-output pairs (supervised), inputs alone (unsupervised), or state-action-reward trajectories (RL)—and require that such a learnt function generalises well to unseen inputs, has reasonable accuracy, and is not especially costly to compute. Of course, given that we lack *a priori* knowledge of the function's true structure or it may lack a closed-form expression, we must use some learnable *approximation*. Such a learnt function is then defined by

a fixed structure (e.g., an NN's architecture) and a *parameter set* $\boldsymbol{\theta}$, which is typically a collection of real numbers. Of particular importance is that the function approximations presented here (and in general) are differentiable with respect to their parameters, which allows for $\boldsymbol{\theta}$ to be trained as we discuss through section 3.3.

Considering the deployment environment we're mainly interested in—PDP hardware—making the right choice of algorithm and the used data format (section 3.4) is crucial. For instance, more complex functions (deeper NNs) have higher *model capacity*, and are capable of learning more complex transformations, but often have a higher cost to use in inference: in either the number of required parameters, or the amount of arithmetic operations needed to produce an output. Accordingly, they can be less well suited for PDP execution. This also correlates with the cost of computing the gradient we'd use to move $\boldsymbol{\theta}$ to a more optimal set of values—in NNs this is more expensive than inference alone, while with simpler linear schemes like tile coding the gradient is acquired for free during the forward pass. Model capacity, inference cost, and learning cost then weigh against the finite resources of PDP hardware. We must then consider several function approximations according to how we want to integrate the ML component of a larger DDN system: online or offline, and in-NIC/PDP or on commodity hardware.

I discuss here some forms of function approximation which are pertinent to this thesis, and to DDN and ML/RL in general. I introduce and explain *linear tile coding*, a simple and rather interpretable scheme which appears throughout more classical RL research, as well as providing an overview of common variations on NNs. While tile coding has been superseded in modern RL works by DNNs, I use it as the main function approximation scheme in chapter 4 for its computational simplicity in both inference and training. Crucially, it plays a key role in the design of OPaL in chapter 5, where it underpins the task of bringing RL to PDP NICs due to their particular constraints. NNs are presented due to their widespread adoption and hence deep relevance to DDN on the whole. Covering their basics also arms us with knowledge of their hardware and software needs for inference and learning—and thus, their suitability to online and offline learning in PDP hardware. In addition, they are used for flow classification in chapter 6 to make good use of the data reduction provided by in-network histograms.

3.2.1 Linear Tile Coding

Tile coding (Sutton & Barto, 2018, pp. 217–221) is a form of feature representation which converts input state s into a sparse boolean feature vector $\mathbf{x}(s)$. A tile-coded representation defines a collection of *tile sets*—each a d-dimensional subset of input state. Each tile set then comprises a set of *tilings*, overlapping d-dimensional grids with different offsets. Overall, this may be combined with an always-on *bias tile*, which is a global estimate that can

contain a reasonable starting point for unlearnt parts of the state space. To compute an output value, input state is checked against every tile to identify which grid cell is activated (or *hit*)—each tile hit corresponds to an entry of $\mathbf{x}(s)$ which is set to 1. As each tiling admits exactly 1 hit, $\mathbf{x}(s)$ is a sparse boolean vector of fixed Hamming weight (i.e., equal to the total number of tilings). The parameter vector $\boldsymbol{\theta}$ then assigns a value to each tile, allowing us to compute the output $f_{tile}(s, \boldsymbol{\theta})$:

$$f_{tile}(s, \boldsymbol{\theta}) = \boldsymbol{\theta}^\top \mathbf{x}(s) \tag{3.1}$$

As all values of $\mathbf{x}(s)$ are boolean, this is equivalent to simply summing the values attached to each hit tile. Moreover, the parameter gradient ($\nabla_{\boldsymbol{\theta}}$) at any point is *computed during the forward pass*:

$$\nabla_{\boldsymbol{\theta}} f_{tile}(s, \boldsymbol{\theta}) = \mathbf{x}(s) \tag{3.2}$$

As a result, RL policy updates are simple to perform—i.e., no extra work is required to compute a policy gradient—so long as it is feasible to cache $\mathbf{x}(s)$, then online updates may be made fairly cheaply, perhaps suiting SmartNICs or similar devices.

Algorithm 1 describes the basic procedure for a single tiling. Generally, tile sets cover overlapping areas by varying the maxima and minima of the tiling to model offsets applied to a fixed-size grid. The number of tiles in any one tiling is easy to know, as we only need to take the product of all entries of *tiles_on_dim*—or return 1 for a bias tiling—and global starting indices into $\boldsymbol{\theta}$ can be precached for each tiling. Figure 3.5a then demonstrates the process for a 2-dimensional state space in an RL context, i.e., defining the state-action value $\hat{\mathrm{q}}(s, a, \boldsymbol{\theta}) = f_{tile}(s, \boldsymbol{\theta})[a]$.[23] It should be apparent that the numbers of tiling sets and the degree of subdivision along each dimension allow a designer to control feature resolution and generalisation. To capture combinatorial effects or represent an input on multiple scales we may combine codings by concatenating individual feature vectors. For instance, different tiling sets may choose the same dimensions with different tile widths, or consider each feature both separately and combined with some covariant property.

[23] It should be noted that the more RL-specific $\mathbf{x}(s, a)$—i.e., approximating the value of a state-action pair—is virtually interchangeable with this representation. We can simply store the values for all actions together, as they are typically all required in value-based RL—if specific tiles are needed to derive the theory or update the representation, we just assign a tile to each action *after* evaluating $\mathbf{x}(s)$.

In an RL context, as in figs. 3.5b and 3.6, this coding strategy is well optimised for discrete actions. This allows a particularly efficient vectorised implementation of the policy and update rules by storing a vector of action values for each tile. Summing the weight vectors from all activated tiles as described, this requires $|\mathscr{A}|(n_{tilings} - 1)$ floating point additions per decision for an action set $\mathscr{A}$. In particular, hit tiles are amenable to representation as an array of indices, s_{list}. This means that we need only perform $n_{tilings} + 2$ additions and 2 multiplications per model update when combined with Sarsa (section 3.3.3):

$$\boldsymbol{\theta}_{t+1}[i][a_t] = \boldsymbol{\theta}_t[i][a_t] + \alpha\delta_t, \forall i \in s_{list}. \tag{3.3}$$

Algorithm 1: Tile coding, for a single uniform grid tiling.

input : A *state* vector and *tiling*. Each tiling has a list of dimension indices (*dims*), minimum and maximum values for each dimension (*mins, maxes*), and a count of tiles for each (*tiles_on_dim*). These lists are empty for a bias tiling.

output : The index of the tile hit *within this tiling*.

```
fn TileCode state, tiling
    let scale: u64 = 1;
    let local_tile: u64 = 0;
    forall (i, dim_idx) in tiling.dims.enumerate() do
        let min: f64 = tiling.mins[i];
        let max: f64 = tiling.maxes[i];
        let n_tiles: u64 = tiling.tiles_on_dim[i];
        let width: f64 = (max−min) / n_tiles;
        let local_hit: u64 = state[dim_idx].clamp(min, max) / width;
        let scaled_hit: u64 = local_hit.max(n_tiles−1) × scale;
        local_tile ← local_tile + scaled_hit;
        width_product ← width_product × n_tiles;
    return local_tile;
```

If desired we may define a state space with an arbitrary number of tiles per dimension (higher-resolution, lower generalisation), yet having constant-size state vectors and constant action computation cost scaling in $\mathcal{O}(n_{tilings})$. Beyond this, we need not store action values for tiles which have not yet been visited, conserving memory. A caveat of tile coding remains, in that the value of α must be reduced according to the number of tilings to prevent divergence at the expense of slower learning ($\alpha \leftarrow \alpha / n_{tilings}$).

In the wider PDP context, the basic tile coding algorithm is simple enough that it is a worthwhile candidate for representing policies in some classes of PDP hardware, as I investigate in chapter 5. In particular, the fact that gradients are acquired *during inference* might be used to make online learning in resource-constrained PDP environments more feasible by saving on valuable compute. A downside is that we store one model parameter per tile, which can scale poorly to larger models—in exchange, we use no non-linear operations. Although I don't examine it in this thesis, it should be possible to implement or even accelerate their forward pass using MATs by mapping 'stripes' of tiles in each dimension to a range match—although it's unlikely that we could extract any online learning capability in RMT-like architectures.

3.2.2 Neural Networks

Neural Networks (NNs) map an input vector to an output vector via a mathematical graph of neurons. Each neuron takes a weighted sum over a set of

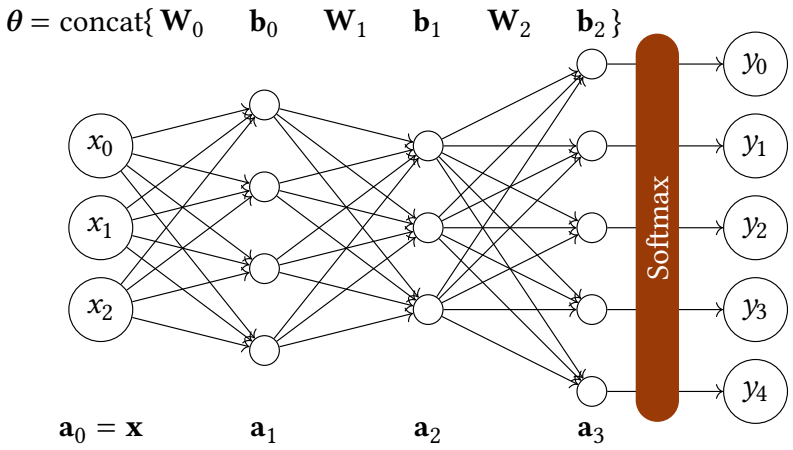


Figure 3.2: NNs arrange neurons into layers, which allows all activations in each layer to be calculated by a single affine transformation followed by an elementwise application of some non-linear function f. I.e., in this fully-connected network $\mathbf{a}_i = \text{map}(f, \mathbf{W}_i\mathbf{a}_{i-1} + \mathbf{b}_i)$. By doing so, they can take advantage of commodity GPUs (which excel at linear algebra) or more specialised TPUs.

inputs plus its own *bias* value, and uses this as the input to a non-linear function, producing a single output value (fig. 3.1). This non-linear (or piecewise-linear) function, e.g., (leaky) ReLU (Maas *et al.*, 2013; Nair & Hinton, 2010), tanh, or the sigmoid function expresses the idea of a sufficiently large input 'activating' the neuron, and prevents chains of neurons from being expressed as a single linear transformation (allowing greater model capacity). As in fig. 3.2, this graph of neurons then progresses from transformations to the *output layer*, towards processing of the outputs of intermediate neurons (*hidden layers*), before terminating in a final vector of output values. The parameter set $\boldsymbol{\theta}$ is then the concatenation of all edge weights and biases which describe the network. This compute graph can contain other, non-neuron operations so long as they are differentiable: it is typical that in classifiers or RL systems the last layer of neurons goes through a softmax function to be converted into a valid probability distribution. Crucially, any individual neuron is differentiable in $\boldsymbol{\theta}$, and by applying the chain rule over the full graph we can compute the entire NN's parameter gradient through the backpropagation algorithm (Goodfellow *et al.*, 2016, pp. 197–217). This includes a forward pass as in inference, coupled with a more expensive (by a factor of ~2 ×) backward pass. Some input data, such as images or fixed-length time-series sequences, often contain clear structural features that must be exploited to learn a function. *Convolutional Neural Networks* (CNNs) (LeCun, 1989) capture these dynamics between adjacent values in a layer by learning a convolution filter.

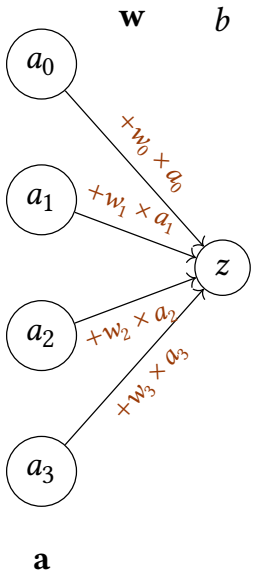


Figure 3.1: A single neuron in an NN takes a weighted sum over its vector of inputs $\mathbf{a}$ (according to $\mathbf{w}$) and its own bias b, and produces an output using some non-linear f. Thus, it produces an output $z = f(b + \mathbf{a} \cdot \mathbf{w})$, where $\mathbf{w}$ and b are trainable per neuron.

Variable-size input data can also be handled by NNs. *Graph Neural Networks* (GNNs) (Kipf & Welling, 2017) attach state to each vertex of an input graph, which are all processed by the same NN architecture and weights. Each vertex's weight is then modified by a message-passing-like model: a transform

is applied over the aggregate of a node's intermediate vector and those of all its neighbours, producing a final embedding vector for each node. Further modifications, e.g., Edge-GNNs (Mirhoseini *et al.*, 2021), exist to include edge labels in the formulation or offer alternate aggregation strategies. In the case of variable length sequences of data (i.e., an obvious time-series use case such as audio transcoding), *Recurrent Neural Networks* (RNNs) (Rumelhart *et al.*, 1986) capture temporal properties of the input. The main way in which they differ is that they feed neuron state from the previous input into the calculation being applied to the current datum, similar to an infinite impulse response filter. *Long Short-Term Memory* (LSTM) (Hochreiter & Schmidhuber, 1997) units include extra gates to control how hidden state is held between timesteps element-wise (input, output and forget gates), whose parameters are also trainable. *Gated Recurrent Units* (GRUs) (Cho *et al.*, 2014) remove the output gate (passing on only the hidden state), and as a result are competitive with fewer weights to learn.

Returning again to the PDP context, NNs are useful because they have both a high model capacity and are quite compactly represented by all parameters in $\boldsymbol{\theta}$—they encode a *transformation* of input data rather than a look-up table. They can thus require less memory to store a policy than tiles might need, at the same count of input dimensions. However, not only is their gradient computation more expensive than the forward pass, their training requires many passes (or epochs) over a large training corpus which should be held at or near the agent under training—likely ruling out their use in *online* learning in PDP hardware. They can be well-represented in RMT hardware and SmartNICs (section 2.4.4), depending on data format (section 3.4), but at the extremes of making inference quick we lose the ability to even modify the policy incrementally. It should be obvious that more complex methods outlined here (GNNs, LSTMs, RNNs) are more broadly unsuitable due to their iterative compute models, outside of the specialised architectures we've discussed earlier in section 2.2.4. Similarly, they are generally understood to be more costly still to train, again ruling out online learning in PDP hardware.

3.3 Learning an approximation

Having introduced function approximation schemes of relevance to this thesis (and the most prominent in DDN at present), we now turn our attention to how these approximators are trained in practice. Due to its relevance in DNN and DRL, this comprises a brief discussion of gradient descent techniques and stochastic optimisers, followed by a more in-depth introduction to and overview of the field of RL. This allows us to comment on these methods' (and key variations') suitability to network control problems and different execution environments (primarily PDP hardware) as required by chapter 5. Tying these back to the realities of DDN, many of these tech-

niques and advances are likely feasible in a PDP context with small datasets and no minibatches, such as *Stochastic Gradient Descent* (SGD) and its variants, but their use is left to future work. Finally, this allows us to consider various problems in deployment and system design where we must expand or diverge from baseline algorithms—(a)synchrony, exploration, and dynamic learning—as needed by applications (chapters 4 and 6) or PDP co-designed algorithms (chapter 5).

We can begin by assuming that our approximate function is backed by some *parameter vector* $\boldsymbol{\theta}$, such that $\hat{\boldsymbol{y}} = f(\boldsymbol{x} \mid \boldsymbol{\theta})$—an input vector $\boldsymbol{x}$ produces some output $\hat{\boldsymbol{y}}$, and ideally after training $\hat{\boldsymbol{y}} \approx \boldsymbol{y}$ across all input $\boldsymbol{x}$. For all intents and purposes, we can understand $\boldsymbol{\theta}$ as a large block of real numbers residing in RAM, and that we update this block over time. We may not have access to a ground-truth $\boldsymbol{y}$ (i.e., in the RL case), but we generally assume that there does exist a best-fitting or optimal output. In case we do have the true $\boldsymbol{y}$ values, we may refer to the complete set of training inputs and ground-truth outputs as $\boldsymbol{X}$. Crucially, we require that f is differentiable with respect to $\boldsymbol{\theta}$: if we define some scalar performance metric J which assesses the quality of an output $\hat{\boldsymbol{y}}$, then $\nabla_{\boldsymbol{\theta}} J$ offers a direction in $\boldsymbol{\theta}$.[24] This output gradient is simply a vector which represents the direction in parameter-space which would have the largest *increase* in the value of J if followed. The approaches presented in section 3.2 are differentiable as required.

[24] In the definition of J, we naturally expand occurrences of $\hat{\boldsymbol{y}}$ in terms of f, and thus $\boldsymbol{\theta}$. Additionally, J may be defined over any subset (or all) of the training set $\boldsymbol{X}$.

3.3.1 Gradient descent

Most supervised ML problems are defined in terms of a *loss function* L, such that differences between our output $\hat{\boldsymbol{y}}$ and $\boldsymbol{y}$ values are penalised. Some examples in use today include the mean absolute and mean squared errors (ℓ_1 and ℓ_2 loss) in regression tasks, or (categorical) cross-entropy loss in classification. Naturally, these are differentiable, but as we aim to *minimise* loss we then *subtract* the gradient from $\boldsymbol{\theta}$; this concept underlies *gradient descent*. Gradient descent is the iterative process of optimising our parameter vector $\boldsymbol{\theta}$, using all input data and labels $\boldsymbol{X}$ at each iteration:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}_t} L\left(\boldsymbol{X} \mid \boldsymbol{\theta}_t\right) \tag{3.4}$$

Every iteration, we move the parameter vector a small step in the direction which would optimise its overall performance; this is the learning rate hyperparameter $\alpha \in (0, 1)$. However, re-evaluating the loss function over all input data becomes intractable in the case that either the function is expensive to compute on a moderately-sized dataset, or the dataset itself is monstrously large. Both are often the case in DNN training. It is for this reason that SGD and related algorithms are typically employed in ML model training, particularly for DNNs. SGD modifies the above formula such that individual, randomly chosen samples (or larger minibatches) are used as the input for the loss function rather than the entire dataset. This only approx-

imates the loss gradient at $\boldsymbol{\theta}$ on the input data, but in practice this is key for the training of modern ML techniques—α may be reduced over time and the dataset reshuffled as necessary until convergence.

While SGD provides the theoretical underpinning for more efficient ML training, in practice the difficulty of choosing α with regard to different parameter sensitivity has led to more adaptive methods. These also introduce the notion of per-parameter learning rates. *AdaGrad* (Duchi *et al.*, 2011) gives each parameter its own learning rate α, divided by the square root of the sum of squared gradient values (i.e., large derivatives in a parameter lead to a greater reduction in learning rate). *RMSProp* (Tieleman & Hinton, 2012) converts this accumulation into an exponentially-weighted moving average, better handling the non-convex loss behaviour seen in DNN training. *Adam* (Kingma & Ba, 2014) includes momentum terms (to carry a general direction in gradient across several steps) and additional bias correction, giving advantages in the early stages of training.

It should be noted that the above discussion is a very cursory treatment of the subject, primarily intended to contrast the in-depth discussion of RL methods in the remainder of the section. Similarly, learning-centric modifications to loss functions such as regularisation terms or function-specific regularisation strategies (to mitigate overfitting) are also out of scope. For more details, readers should refer to more specialised texts such as Bottou *et al.* (2018) and Goodfellow *et al.* (2016).

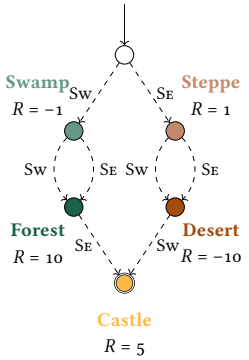


Figure 3.3: A toy example showing how the *delayed rewards* which an RL agent learns to maximise. Consider a traveller journeying towards a castle in the south, with R being some aggregate of food, water and rest. Learning only to optimise *immediate* rewards attached to actions would lead our traveller to choose an overall worse path—going thirsty in the desert!

3.3.2 Reinforcement learning

When we aim to optimise for decisions and estimations in classification, clustering, and regression, it suffices to apply gradient descent and similar optimisers to minimise this loss function alone. How might things differ when we want to design some *agent*—an actor who sees and acts on a system for many timesteps—to interact with the world? Consider a toy example in fig. 3.3: an oracle with global knowledge knows that a hypothetical traveller would best be served by going through the swamp, even if it appears the worse of two choices. Yet if we applied simple input classification to choose our path at each turn, we would always act poorly without some way of propagating knowledge of later choices' value. It becomes more complex to collect training data once we realise that our actions move the world forward and change it (thus we may not be able to rewind a simulation to explore alternative choices), and that we might need to explore apparently suboptimal choices for some time to be better off in the long-term.

Reinforcement Learning (RL) algorithms are methods of training such an agent to choose an optimal sequence of actions in pursuit of a given task (Sutton & Barto, 2018), and neatly encode these training- and run-time notions of *exploration* and *exploitation*. An agent is typically defined by its *policy* π, which chooses an action in response to an input state. These powerful

techniques effectively use a *reward* metric to learn a state-action mapping without any explicit model of the system they're learning to control—even when reward signals are sparse or come with some delay after an initial choice as fig. 3.3 exemplifies.

To achieve the goals described above, we first make the assumption that the task we're attempting to solve is structured as an MDP. In an MDP, we break the world or target problem down into a set of *states* ($\mathcal{S}$), *actions* ($\mathcal{A}$), and *reward measurements* ($\mathbb{R}$). To relate this to computer networks, an example state space would be a vector of buffer occupancies in a switch, an action space would be the priority to assign some new flow which has arrived, and the reward might be the proportion of finished flows which achieved comfortably low FCTs. We then consider our sequence of interactions in discrete *timesteps*—at the present time t, an agent observes $s_t \in \mathcal{S}$, chooses some $a_t \in \mathcal{A}$, and then observes their change in the world as a new state $s_{t+1} \in \mathcal{S}$ and a reward measure $r_{t+1} \in \mathbb{R}$. As required, we can qualify these further, e.g., the set of actions we can take from a state s may be limited to $A_s \subseteq \mathcal{A}$. Acting optimally then consists of maximising the sum of rewards over some time horizon. This is captured up to an infinite horizon by the concept of the *expected discounted return*:

$$\mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t\right] \tag{3.5}$$

where the *discount* $\gamma \in [0, 1]$ controls how "forward-planning" an agent may be[25], always choosing $a_t \sim \pi(a_t \mid s_t, \boldsymbol{\theta}_t)$. While dense, this formalism essentially captures and describes the performance of our agent over all possible run-lengths from all starting states (i.e., over all *episodes*), and in general this is what RL algorithms are designed to maximise.

[25] Setting $\gamma = 0$ defines a 'myopic' agent, where no intermediate loss of reward is allowed. Choosing $\gamma = 1$ will prioritise future rewards, with no concern over how long it may take to achieve those rewards. This hyperparameter features in many RL algorithms; in practice, we choose values closer to 1.

MDPs assume that there are stationary, well-defined (though stochastic) transitions between these states. For any $s, s' \in \mathcal{S}, a \in A_s$, an MDP is defined by a state transition function, $\mathrm{P_a}(s, s')$ which returns the probability that we will land in s' after taking action a in state s, and a function returning the *expected reward* $\mathrm{R_a}(s, s')$ for each valid transition. If we have all this information, then we can apply the Bellman equation (Bellman, 1957), a dynamic programming algorithm, to solve for an optimal policy. This allows us to assign a *value* $v_\pi(s)$ to each state (the expected return over all choices in the current state), and $q_\pi(s, a)$ to each action, and then choose maximal-value actions at each timestep. In real-world scenarios however, we usually lack this knowledge; this either requires involved modelling, or is rendered intractable by a continuous or combinatorially large state space.

Like most ML methods, modern RL algorithms use gradient information to update the parameters used to approximate a function. How RL differs is that it aims to learn an optimal policy without any knowledge of the MDP apart from its state and action space—*model-free*. Knowledge of the *trajectories* followed by agents (i.e., state-action-reward tuples at all timesteps) is

then used to compute target values and adjustments for the parameter set $\boldsymbol{\theta}$ for any function approximator. By requiring that our policy approximation $\pi(a \mid s, \boldsymbol{\theta})$ is differentiable, RL works in tandem with any of the function approximation approaches described in section 3.2.[26]

[26] Note the policy π does not require any knowledge of rewards, except when training. This allows for RL agents to be trained using simulated systems or data, and then deployed in an environment where they cannot access this knowledge due to sampling cost. If these measures are present in deployment, then RL can learn online.

3.3.3 Demonstrating RL: Sarsa

While many RL algorithms have been developed (each of which making quite different choices on how to apply trajectory data), it is likely most helpful to start with a concrete point in the design space *before* listing their full variety. Doubly so here, as this particular algorithm—single-step, semi-gradient Sarsa (Sutton & Barto, 2018, pp. 217–221)—underpins chapters 4 and 5. The Sarsa algorithm considers only one transition at a time: a pair of states s_t, s_{t+1}, their accompanying actions a_t, a_{t+1}, and the reward r_{t+1} accompanying s_{t+1}. This is why it is defined as *single-step*, and as such it does not require a completed trajectory for learning.

As Sarsa is a *value-based* method, an agent operates by defining an approximate *value* function $\hat{\mathrm{q}}(s, a, \boldsymbol{\theta})$ to each action a it can take from s, typically choosing the action with the highest value. Note that $\hat{\mathrm{q}}(\cdot)$ is completely arbitrary, and may be any differentiable approximator. We can then update $\boldsymbol{\theta}$ over time as follows:

$$\delta_t = r_{t+1} + \gamma \hat{\mathrm{q}}(s_{t+1}, a_{t+1}, \boldsymbol{\theta}_t) - \hat{\mathrm{q}}(s_t, a_t, \boldsymbol{\theta}_t), \tag{3.6a}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \delta_t \nabla \hat{\mathrm{q}}(s_t, a_t, \boldsymbol{\theta}_t), \tag{3.6b}$$

where δ_t is known as the *Temporal-Difference* (TD) error. $\alpha \in (0, 1)$ defines the learning rate (governing the policy stability), with γ defined as in the MDP formulation.

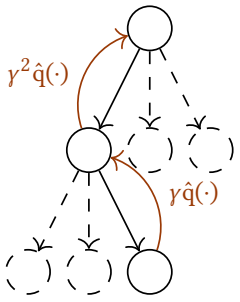
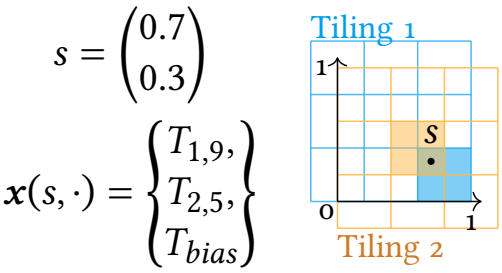


Figure 3.4: A simplified view of how action values propagate back through visited states: because a state's value includes some portion γ of its children's values, at convergence it includes γ^2 of its grandchildren, and so on. A limitation of the single-step family is that they must visit the same transitions several times for earlier states to learn about their successors.

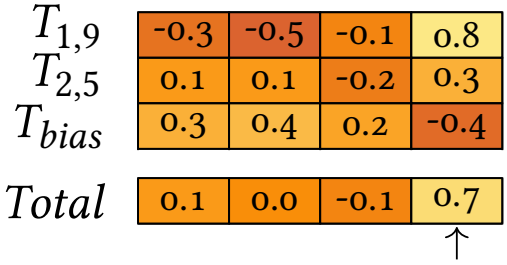
In essence, at each timestep the policy parameters ($\boldsymbol{\theta}$) are increased along the gradient ($\nabla \hat{\mathrm{q}}(\cdot)$) using a fixed learning rate (α) and a computed adjustment (δ_t). This adjustment is equal to the difference between the chosen action a's value in state s and the reward received ($r_{t+1} - \hat{\mathrm{q}}(s_t, a_t, \boldsymbol{\theta})$), plus some part of the *next* action's value ($\gamma \hat{\mathrm{q}}(s_{t+1}, a_{t+1}, \boldsymbol{\theta})$). Naturally, as state transitions are visited and revisited during training, the value of later choices can propagate down the tree of states, as shown by fig. 3.4. To see how this combines with a simple linear function approximation, consider figs. 3.5 and 3.6.

3.3.4 The RL algorithm design space

While the above introduction to Sarsa is a clear way to intuit the key ideas underpinning RL, it is a very specific point within a remarkable design space. For context, other algorithms may employ separate state-value approximations, use the entirety of an agent's trace, or be tailored to characteristics of



(a) Tile Coding



(b) Value Estimation and Action Selection

Figure 3.5: An example of tile coding for 2-dimensional state and 4 actions, using 2 tilings, 3 tiles per dimension, and a bias tile. All components of s are clamped to $[0, 1]$. For simplicity, I denote $\boldsymbol{x}(s, \cdot)$ as a list of indices and represent the values of all actions at each tile with a vector. (a) The state s activates the bias tile and exactly one tile in each tiling. (b) The action values of active tiles are summed to produce the current value estimate for each action available in s—for this state, local knowledge ensures that action 4 is chosen by the greedy policy despite typically being a poor choice elsewhere.

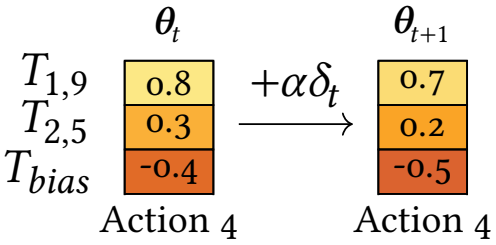


Figure 3.6: The update step for fig. 3.5, given an observed TD error $\delta_t = -0.2$ (indicating a lower observed reward than the expected long-term value of 0.7) and $\alpha = 0.5$. Action 4's value is thus reduced in the tiles associated with state s, but remains the most likely choice; the negative δ_t may have arisen from noise in the reward signal. For illustrative purposes, I choose an unrealistically high α (typically, $\alpha \leq 0.05$ is a more reasonable choice).

the policy approximator (e.g., how neural networks benefit from batching). Algorithms tend to differ in some key ways:

Trace length. Single-step methods like the above can be generalised to *n-step methods* by including further discounted reward measurements (at increased runtime memory cost), as in *A3C* (Mnih *et al.*, 2016). *Monte Carlo methods* such as *REINFORCE* (Williams, 1992) carry this to its logical extreme, updating every transition in a trace using the known return for an episode. This solves the issue of 'repeat visits' hinted at in fig. 3.4, at the cost of storing entire trajectories. Moreover, these can be tricky to delineate into clear episodes in some online tasks. Consider also *eligibility trace* methods such as TD(λ) (Tesauro, 1995), which propagate value backwards through the MDP by including some portion $\lambda \in [0, 1]$ of the last timestep's gradient alongside the current (i.e., having some decaying part of *every* prior state's gradient).

The role of values in a policy. Sarsa (and many other algorithms) follow a *value-based* approach: the role of the function approximator is to compute and learn the value for each action, and then action selection chooses based on these values. This design, however, prevents continuous control (i.e., $a_t \in \mathbb{R}$). The dominating paradigm of late has been *policy gradient methods*, which impose no requirement on the policy's output format—given that a differentiable performance metric in $\boldsymbol{\theta}$ is known. This allows easier expression of many system designs, such as having mixed real-valued and discrete action components. The development of *DPG* (Silver *et al.*, 2014) has been a key driver in ensuring their suitability for continuous problems. *Actor-critic* algorithms are a considerable subset of this which also learn separate a value estimate for the current state to provide this performance measure. Going further still, *direct policy search* approaches such as those inspired by or using ES eschew gradient computation to apply randomised modifications directly to the policy.

While the high-level conceptual directions and differences between these algorithms are interesting, we should return to what they imply for online deployment in PDP hardware. Additional trace length means that we must dedicate extra runtime memory *per trace* for either state-action-reward tuples and values—high-speed memory of course being in short supply in this environment. However, even if we don't cache gradient values themselves[27] the computational cost does not substantially increase beyond including additional discounted value pairs, meaning that there may be an acceptable tradeoff here. Policy gradient methods like actor-critic algorithms may prove trickier even with discrete actions, as they require additional compute and storage to maintain *two or more* parameter sets which may overlap or be disjoint. ES methods (Chrabaszcz *et al.*, 2018; Salimans *et al.*,

[27] This is the case in the *ParSa* algorithm introduced in chapter 5, where recomputing the gradient is faster. We only require one gradient measure per update still—in an n-step method, this is the gradient for the state-action pair n steps ago.

2017) may instead be a comfortable fit for policies with fewer parameters. By adding a small amount of random noise over $\boldsymbol{\theta}$ and aggregating successful policies, these algorithms are devoid of *any* gradient computation and admit efficient communication schemes.

3.3.5 RL use considerations

Exploration vs. exploitation in practice Consider again the dilemma presented in fig. 3.3: discovery of an optimal policy relies on either global knowledge, or *exploration* of a suboptimal portion of the state-space. RL agents are typically initialised with either zeroed or random policy parameters, but we cannot expect that for larger state spaces this produces *meaningful* exploration. It may well be the case that following the current optimal policy up to some uncertain state and then exploring can provided targeted and useful samples, whereas randomised parameters are something more akin to a random walk for all timesteps spent in early episodes.

To encourage exploration in discrete action spaces, we typically introduce some randomness into action selection. *ϵ-greedy methods* pick another action at random with probability ϵ—typically annealing the value of ϵ towards zero over some number of timesteps or training episodes. Meanwhile, the simplest way to achieve this in many NNs is to use the outputs of a softmax (Luce, 1959) layer as a probability distribution over actions, particularly if starting from randomly initialised parameters. Boltzmann and Max-Boltzmann action selection (Wiering, 1999, p. 73) constitute variations of these schemes which control the concentration of action probabilities. Active estimation of the degree of exploration has also attracted healthy degrees of interest (particularly with regard to evolving or non-stationary problems): VDBE (Tokic, 2010; Tokic & Palm, 2011) uses changes in the TD error to control ϵ over time, while Tokic and Palm (2012) train a continuous NN-backed agent to control exploration parameters.

In the case of continuous RL action spaces, an initial consensus in the wake of the DDPG algorithm (Lillicrap *et al.*, 2016) was to make use of Ornstein-Uhlenbeck processes (Uhlenbeck & Ornstein, 1930) to directly inject noise in the action space. However, more recent ablation studies have shown that this offers no tangible benefits over Gaussian noise (Barth-Maron *et al.*, 2018; Fujimoto *et al.*, 2018).

RL in asynchronous data-driven networking Automatic, adaptive control requires accurate, recent state to make optimal decisions and to act in a timely manner. Action execution, computation and training have real costs, which have been shown to negatively affect the performance of asynchronous RL systems (Travnik *et al.*, 2018). Hence, DDN applications are profoundly affected, as they must often reckon with inference times which are significant compared to the systems they control. As it stands, state measurement and

policy execution require additional hardware and infrastructure, increasing delays and costing rack-space in the data centres or networks where we aim to deploy DDN.

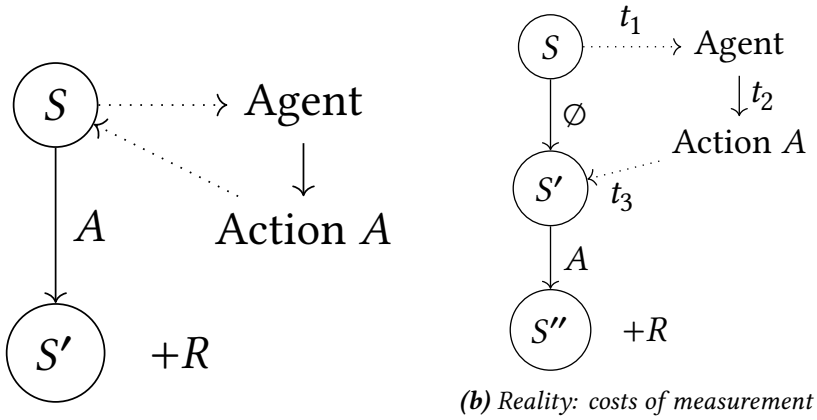
There remains some degree of divergence between the theory and implementation of RL agents when it comes to accounting for the above costs. Consider fig. 3.7: the traditional formulation of an MDP assumes that an agent receives a new view of the world's state at fixed time intervals, and then decides upon and executes an action instantly. The reality is that state information takes time to traverse the network, service times are offset by how quickly hosts respond to interrupts and deserialise requests, and action preference lists are often computed via expensive policy approximations. Action installation also incurs costs in fields such as network administration, initially to contact the controller and then for those actions to be installed via the control plane.

These delays (and variance thereof) add noise to the state-action mapping being learnt, which has a potent reduction to learning rate and final accuracy, even for simple grid world tasks according to Travnik *et al.* (2018). They in turn show that reordering algorithmic steps can reduce these costs for online single-step algorithms, but that reducing this further requires detailed agent-environment co-design. Achieving this often requires that state measurements are combined or coalesced (K. A. Simpson, Rogers & Pezaros, 2020; Sivakumar *et al.*, 2019) while expensive computations are ongoing. For instance, 'stopping the world' in the algorithmic sense causes significant performance degradation, as inference takes up to 30 ms for Sivakumar *et al.*, or any time-sensitive control problems. In the PDP case, this can be important for a wide variety of reasons; we might be interested in capturing statistics of a controlled system over longer timescales, or we might need to explicitly rate limit requests at switch-scale or line-rate to prevent an agent (and its parent NIC or switch) from being overloaded.

3.4 Numerical representations for embedded ML

ML training and inference work in the domain of real numbers, and thus require a suitable data format for representation of weights, gradients, and values. To consider how to perform inference and learning in different classes of PDP hardware, and thus enable online RL as in chapter 5, we must weigh up the impacts and hardware requirements of suitable data formats.

Floating-point arithmetic is the canonical data format for this purpose, and allows commodity machines and accelerators to approximate real numbers using a fixed-size representation, dividing available bits among a *sign*, *exponent*, and *mantissa*. This captures several important properties, principally *dynamic range* (as the exponent describes the magnitude of a number). For instance, 32 bit floating-point numbers (FP32) use 1 bit, 8 bit and 23 bit to



(a) *Theory: state measurement, action computation, and learning are zero-cost.*

(b) *Reality: costs of measurement and action lead to* state drift*—over a time delay $t_1 + t_2 + t_3$, inaction transforms state S into S'.*

Figure 3.7: One issue which arises when we aim to introduce RL into temporally fine-grained environments is that the MDP formulation can fail to capture how state drifts during computation. Input states may take some time to acquire and transmit to an agent over the network (t_1), the function approximator itself has an associated computational cost for inference (t_2), and enacting said action can involve network latency *and* expensive table preprocessing in the case of hardware P4 implementations (t_3). As a result, the system actually undergoes the transition $(S', A) \rightarrow (S'')$, introducing noise or variance into the value function being learnt.

store each component, which is sometimes known as a 1-8-23 representation.

However, there are concrete reasons to consider other data formats, particularly on more resource-constrained environments. Quantisation and alternative data formats have been suggested to make ML inference feasible on resource- and power-limited platforms, work around hardware constraints, or compute faster and more efficiently. Although individual floating-point operations, as compared to integers, have effectively equal latency and reciprocal throughput on modern x86 hardware (Fog, 2021)[28], FPUs still require additional chip area and power. Naturally, chip designers don't want to fabricate or plan around unnecessary FUs: for instance, (programmable) network hardware and ASICs require only basic integer arithmetic. This is not the only reason to be interested in alternative data formats; by reducing the size of any individual number from 32 bit to 16 bit or 8 bit, we reduce the size of parameter sets and input vectors by two to four times. This reduces the range of numbers we can express (in both magnitude and precision), but can reduce inference latency and memory cost for the benefit of both commodity machines and dedicated accelerators. Luckily, the task of bringing ML models to resource-constrained environments without these capabilities is well-studied, and in general the effect on accuracy is small in spite of the introduced quantisation noise.

[28] This leaves aside the performance gains offered by SIMD vectorisation, which is a trickier topic.

To deploy ML models to PDP hardware, the question of what data format

to use is one we *must* ask ourselves. Simply put, this is a necessity due to the general (and expected!) lack of floating-point support in PDP hardware of all sizes. What might be less obvious is that we have different needs depending on whether the main task we're interested in is inference—i.e., per-packet classification to detect attack traffic—or in-situ learning of a function or policy (such as adapting a learnt AQM scheme to account for a new traffic class). The speedup and memory savings we can gain from any choice are dependent on the target environment (e.g., between PDP switches, NPUs, or host machines), as is our ability to compute and manipulate policy gradients. We consider here the value and use of many fixed-point, floating-point, and binarised representations for online and offline ML in different execution environments across the network.

3.4.1 Floating-point

Floating-point formats are the *de facto* way to represent real numbers, but for the reasons discussed above cannot be included in most PDP use cases. However, they may be readily used on host machines (i.e., in longer-term control plane inference tasks), in some environments such as FPGAs where we can deliberately include the needed FPUs, or dedicated accelerators such as *BrainWave* (Fowers *et al.*, 2018). Finally, they are most obviously suited to the actual task of learning a function—their dynamic range trivially allows us to represent input values and capture small gradient values in each parameter.

While there are standardised floating-point forms designed to target mobile and weaker hardware, such as *half-precision* (FP16, 16 bit, 1-5-10) and *mini-float* (FP8, 8 bit, 1-4-3), these fail to be effective in some ML use cases. At the same time, a key factor in FPU chip cost is the size of the *mantissa*—which has been observed to have a quadratic scaling effect on area in TPU development (S. Wang & Kanwar, 2019). Accordingly, allocating more bits to the *exponent* can allow for more cores and FUs in the same area, or reduce power draw. *bfloat16* (S. Wang & Kanwar, 2019) is a 16 bit format employed in Google TPUs (S. Xie *et al.*, 2018) and modern Intel Xeon server CPUs (Intel, 2021a) among other devices. It matches the dynamic range of 32 bit floats (1-8-7), better expressing the smaller end of the dynamic range (e.g., for gradients) while having identical failure modes (subnormal numbers, edge cases) to FP32. *hfp8* (Sun *et al.*, 2019), as an 8 bit format, uses different layouts for the forward (1-4-3) and backward (1-5-2) passes, applying a downward bias of 4 to the exponent in both cases. This allows better expression of small values in general, and even smaller values during gradient computation, at an extra 5 % hardware area cost to support both formats. While this is a fairly indicative summary of high-profile floating-point variants today, it must be said that there are more formats beyond the scope of this introduction (Mach *et al.*, 2020).

3.4.2 Fixed-point and binary

While the above techniques are promising and effective, deployment on environments such as PDP hardware requires further ingenuity. *Fixed-point arithmetic* is an approach which makes it possible to represent real numbers as integers, losing dynamic range as a consequence. The *Qm.n* format expresses a real number using an m bit integer part alongside an n bit fractional part, which allows us to evaluate and update policies using only integer arithmetic on $k = 1 + m + n$ bit numbers, assuming the presence of a sign bit. For instance, the 8 bit number `0b0010_1000` in $Q3.4$ represents the real number 2.5—we can view this as two separate parts $(2 + 8 \times 2^{-4})$, or as one whole in the fractional base (40×2^{-4}). In practice, the entirety of the number is stored as a two's complement number in place of a sign bit, and base conversion (i.e., changing n) requires only bitshifts. When k is known, we can simply refer to the representation as Qn. The most useful part of this scheme is that integer addition and subtraction are *unchanged* for two Q numbers, and conversion of a normal integer requires an n bit left shift. Multiplication and division *by a normal integer* can be performed using the standard ALU operations, while Q numbers need an additional bitshift (pre- & post-op base correction) and temporary expansion into a $2k$ bit register.

Although these tradeoffs seem to predispose fixed-point towards *only* target environments without FPUs, it has still enjoyed application in inference and training. In particular, Courbariaux *et al.* (2015) were able to train Maxout networks without substantial error with k at 19 bit, or 11 bit using dynamic scaling ($m = 5$). In distributed training, fixed-point arithmetic is useful as an intermediate representation for in-NIC gradient aggregation (Lao *et al.*, 2021; Sapio *et al.*, 2021). Scaling down NNs to INT8 requires careful calibration (Migacz, 2017), binning the activations per layer in an FP32 network to choose optimal saturation thresholds. Training INT8 DNNs directly on mobile and *Internet of Things* (IoT) devices has become more possible (Zhou *et al.*, 2021), though this still requires floating point hardware to compute simple compensation terms after moving all tensor operations to fixed-point. To some extent, these can also enable in-NIC DNN inference (Langlet, 2019), and have been used to great effect in the earlier-described *Taurus* (Swamy *et al.*, 2020; 2022) architecture for in-network ML (sections 2.2.4 and 2.4.4).

Binary representations are used to great effect in *Binarised Neural Networks* (BNNs) (Hubara *et al.*, 2016; M. Kim & Smaragdis, 2016; Miyashita *et al.*, 2016; Rastegari *et al.*, 2016); by using 0 to represent to −1, and 1 to represent +1, we may replace Hadamard product operations between tensors with XNOR operations, and when combined with POPCNT instructions we can easily compute the dot product between vectors. This offers bit-parallel execution compatible with almost all ALUs, including PDP hardware. This enables in-NIC execution of NNs: either offloading small portions of fully-connected layers to accelerate inference (Sanvito *et al.*, 2018) or to enable line-rate packet processing in P4-capable PDP hardware (Siracusano & Bi-

fulco, 2018; Siracusano *et al.*, 2020). This is achieved by converting the trained model into MATs—as such, their execution is accelerated by data-plane specific primitives such as TCAM. While there is vested interest in running these complex function approximators in NICs and switches, they are at present trained on commodity x86 machines using real-valued weights and gradients clamped to $[-1, 1]$ (i.e., using tanh). While very effective (and portable) in inference, we cannot represent gradients or partial adjustments of learnt parameters. As such, online training with binarised formats remains out of reach, in spite of their in-NIC suitability.

3.5 Challenges

Leaving security aside for now, there remain key issues in the training and design of DDN-backed systems which we must consider in the evaluation of DDN solutions like those in chapters 4 and 6. I briefly discuss several high-level challenges in the applicability of DDN, and initial in-roads to their solution: issues inherent to data collection and simulation in representative environments; the seeming lack of generality of ML and RL models in the systems domain; and the interpretation and verification of trained function approximators, particularly when given control over such critical infrastructure as the Internet.

3.5.1 Input data and simulation

A key issue in DDN is that training from trace data is inherently fraught with risk. Consider that traces contain fixed sequences of states, and it should be apparent that through their use we cannot model or learn how the controlled system acts in response to change. Traces include the tacit assumption that the model will not change in response to the input, either due to an unforeseen operational mode, or because the users of the system actively change their behaviour. Even given these difficulties, why do some studies still attempt to learn network control in this manner? Network operators are, broadly speaking, unwilling to apply untested and unverified ML or RL models towards production networks; not only because they aim to prevent misconfiguration or collapse, but to uphold contractually enforced *Service-Level Agreements* (SLAs). Overcoming this requires the design and implementation of accurate simulations, which are marred by complex interactions between and across levels of the (ever-evolving) networking stack, particularly at Internet scale (Floyd & Paxson, 2001). Consider a task such as video stream ABR selection: a simulation must consider at the minimum client-side and server-side behaviour (resources, contention, and demand), transport-layer protocol dynamics (handshakes, failure modes, CCAs), and path characteristics (including load balancers) to name but a few. These concerns are neither new nor limited to the field of DDN:

> Here you can see the problem clearly. It is not that simulations are not essential these days, and will be in the near future, but rather it is necessary for the current crop of people, who have had very little experience with reality, to realize they need to know enough so the simulations include all essential details. How will you convince yourself you have not made a mistake somewhere in the vast amount of detail? ... The relevant accuracy and reliability of simulations are a serious problem.
>
> (Hamming, 1997, p. 248)

The main difficulty arises from the fact that it takes not only expert knowledge to model these dynamics but to *consider* them, and while most of these details can be abstracted over it is harder to determine those which will not lead to further surprises in deployment.

Taking advantage of trace data in a more principled way requires insights from *off-policy* RL, such as the use of doubly robust estimators (Bartulovic *et al.*, 2017) or contextual bandits (Lécuyer *et al.*, 2017). These include the derivation or learning of reward models, and importance or inverse-propensity sampling. Even so, in the case of doubly robust estimators Bartulovic *et al.* explain that these may be invalidated by hard-to-model or highly non-linear assumptions. In addition, different policies will invoke different state distributions, and these approaches are incompatible to some extent with non-Markovian problems or non-stationarity.

While it might be easier to cynically connect this goal to the initial wave of dataset-driven ML algorithm applications papers, trace data *can be correctly handled.* For instance, using RL to solve static problem instances derived or cast similarly to NP-Hard optimisation problems does not require simulation or ongoing interaction with a 'real' environment. Although it can be difficult to know ahead of time, it's worth considering whether the problem is adversarial in some way; an ongoing control problem is altogether different from an offline optimisation task. It's unlikely that in an optimised chip floorplanning task, for instance, that target programs will immediately start acting differently—compare this to a network where our actions immediately induce varied modes in CCAs.

3.5.2 Generality

As we can see throughout section 3.1, DDN methods are applicable to a wide variety of problems. However, to claim that these use cases require in-depth agent-environment co-design is a generous understatement—particularly applications of RL, which require very careful consideration to construct a sensible MDP formulation. State and action space definitions have potent effects, are inherently problem-specific, and require domain expertise to define and iterate on.

Recent research aims to investigate whether more general approaches are feasible, by using ML inference to convert input vectors and decisions into a performance metric (Fu *et al.*, 2021)—effectively as a black-box form of 'what-if' analysis. Ideally, there would be no case-specific design elements beyond the chosen input features, offering accurate performance prediction would function for many separate applications. Looking at scheduling, planning, and placement tasks[29], they find that inbuilt optimisations add irreducible variance to the learning problem, even when the task is made as simple as possible. Non-deterministic behaviour (i.e., stochastic load balancing) is for instance an obvious source of noise, but threshold-based behaviours also cause significant discontinuities. Even after diagnosis of these noise sources (negating the desired ease-of-use), input-sensitive tasks still require probabilistic ML methods, which can be harder to reason about or act on.

[29] Generally, the varied parameter which must be optimised is some form of compute allocation, e.g., the number of servers or executors given to a task set.

3.5.3 Interpretability and verification

For all the effort, time and expertise required to craft them, algorithmic or heuristic solutions to network problems have a key advantage over data-driven methods. Because they are so well-specified, it is reasonable for a network operator or expert who has observed some unintended behaviour to derive the root cause, and possibly formulate a fix for the issue. In contrast, ML and DDN models' behaviour is governed almost entirely by an opaque set of parameters ($\boldsymbol{\theta}$, which can contain 10^3–10^9 real numbers), which makes it harder to understand what aspects of input data are being acted on and prioritised. As a side effect, tweaking a model to correct, modify, or improve behaviour is also rendered difficult or impossible. *Interpretability* captures whether a human can reasonably understand why an input or scenario invokes an output or set of behaviours. *Verifiability* describes our ability to prove that a DDN or heuristic solution upholds key properties through, e.g., modelling or closed-form analysis.

Interpretability in general ML has attracted attention as a research goal in its own right. Many classical ML approaches such as SVMs or decision trees offer sensible rule sets or decision boundaries (Molnar, 2021; Molnar *et al.*, 2020), yet NN-based function approximation presents very particular challenges. These comprise repeated high-dimensional linear transforms of input data (e.g., by matrices containing thousands of values), interposed with non-linear activation functions. CNNs and LSTMs complicate this logic further by introducing learnt convolution filters and temporal relationships, respectively. On some data classes such as images, it is possible to visualise learnt feature activations (Olah *et al.*, 2017), which typically manifest as shapes or patterns that make some degree of sense to a human observer. Network state spaces, however, are far less intuitive, so feature activations in NNs are less obviously meaningful and still fail to allow configurability. Tools such as *LIME* (Ribeiro *et al.*, 2016) can reveal the relative importance of each feature in such cases, but it can still require in-depth testing to realise

that (let alone *why*) an agent never chooses some actions in spite of their optimality (Dethise *et al.*, 2019). A DDN-specific remedy is to use teacher-student methods to convert (non-recurrent) NNs into simpler models (Meng *et al.*, 2020). In particular, 'local' decision-making systems (CCAs, ABR selection, TO/TE) are converted into decision trees after applying branch pruning algorithms to keep the model compact enough to be understood. For global decisions (job scheduling, routing, VNF placement) they produce hypergraphs which express which decisions are critical in an optimal solution. In addition to reducing latency and making it clear what sequence of decisions is responsible for an output, this exposes any 'missing classes' in the input and output spaces quite visibly. In response, administrators may add additional training data or modify the decision tree themselves. Unfortunately, the hypergraph representation fails to explain or simplify the logic behind a given decision set (as opposed to the highest-value members of that set), but can allow model co-design, i.e., important features can be allowed to skip several NN layers (having a more direct effect on output).

Verification has a broad set of meanings in ML research, from investigating security properties (i.e., adversarial robustness[30]) to more global guarantees of input and output properties. One promising line of work in this area applies SMT solvers to smaller DNNs for, e.g., safety and liveness constraints on inputs (Katz *et al.*, 2017; 2019). These verification techniques are powerful in that they can guarantee a desired property is upheld (or produce a distinct counterexample where it is not), although recalling that SMT solution is NP-Hard, this comes at a high execution cost. Luckily, most DDN use cases considered by this thesis apply small-to-moderately sized networks, to which such verification is well-suited—moreover, latency and throughput demands incentivise the use of smaller NNs. Extending this towards RL is trickier, given that we must now verify that properties hold over state trajectories of arbitrary length. In addition, the onus now falls on system operators to design suitable state transition functions (i.e., accurate system simulations) to model how a system evolves in response to an action. With these primitives, some degree of DRL verification is possible via bounded model checking (Eliyahu *et al.*, 2021; Kazak *et al.*, 2019) (checking run-lengths of n states from a set of initial states), and k-induction (Amir *et al.*, 2021). In addition to the need to define state transition dynamics using only linear functions, these works also impose strict limits on policy non-determinism and activation functions which can be used.

[30] I discuss verification as an approach towards adversarial example resistance in section 3.6.

It's worth noting that, as with several approaches examined in section 3.1, a hybrid approach can be useful in reducing the impact of both these concerns. By augmenting an algorithmic or heuristic approach with DDN to compute optimal parameter choices, it can be far more reasonable in practice to understand how a system on the whole will behave. Equally, it becomes easier to bound such choices within a safe range as needed.

3.6 Security

ML models introduce particular security issues in their training, use, and how we expose them or their decisions to users. Recall that, in general, their operation is *entirely governed by their parameter set* $\boldsymbol{\theta}$, and that we currently face great difficulty in understanding exactly what transformations or logic they encode. What additional concerns might arise from this? The most obvious challenge is that an attacker might construct input samples which appear to a human to have one label, but produce a strong response in a DDN model for *another label*. We call such inputs *adversarial examples* or *evasion attacks*. Changing focus, the idea of online or *active learning* (Settles, 2010) can seem like a powerful capability to have in the administration of a network for saving operator time. In introducing this, we now need to ask how an attacker might aim to covertly modify our model's behaviour, either to change the label for a handful of samples (e.g., ensure a malware sample always evades inspection), adjust the entire decision surface (e.g., to incur a *Denial of Service* (DoS) or performance degradation by incorrectly handling *all* flows), or to encode an input pattern which always triggers a given output. These behaviours fall under the umbrella of *poisoning attacks*. In tandem, we must also ask whether attackers are able to reverse engineer our model parameters from queries or environmental observation, and the privacy implications of a parameter vector $\boldsymbol{\theta}$ being leaked or extracted—*data extraction* attacks. These classes of attack, interestingly, mostly mirror those that have historically threatened classifiers in the security domain (Barreno *et al.*, 2006).

I introduce in this section the techniques and procedures for undertaking these classes of attack, in addition to defences and present suggestions on how and why they work. It must be stated that the attack and defence surface of ML models is very much subject to the same game of cat-and-mouse as any other security domain, e.g., malware or DDoS design and detection. This field in particular moves very quickly due to the larger reach and impact of DNNs in society as a whole, motivating constant scrutiny by the security community. As a result, any defences listed are certain to have been invalidated by the time this thesis is read; I hope this section at least provides an illustration of the *classes* of input, output and model transforms that have held some promise.

3.6.1 Evasion attacks and adversarial examples

Adversarial examples are input data which have been subtly modified to trick a machine learning model into producing an incorrect output (Papernot, McDaniel, Jha *et al.*, 2016; Papernot *et al.*, 2018). This problem has been known to security experts for a much longer time under the moniker of *evasion attacks* (Barreno *et al.*, 2006). The context for these evasions includes

cases as simple as spam filter avoidance, and as complex as self-modifying and virtualisation-aware malware (Copty *et al.*, 2018). The term does not purely cover ML-based approaches in this context, though there are similarities in the sense that the transformed output must maintain a key property (i.e., it remains a functioning malware payload).

For instance, assume we have in input vector $\mathbf{x}$ with a ground truth label w that the classifier correctly outputs. An attacker wishes to add some *perturbation* $\boldsymbol{\delta}$ such that the adversarial example $\mathbf{x} + \boldsymbol{\delta}$ produces a new output w' from a classifier but still appears to belong to w according to a human observer. They may require that w' is a specific label, or simply that $w \neq w'$. These attacks typically assume a white-box attacker (i.e., one who has direct read access to the ML model's parameters), who is able to use their knowledge of $\boldsymbol{\theta}$ to compute this $\boldsymbol{\delta}$. The data extraction techniques discussed shortly (section 3.6.3) offer more concrete tools for mounting a black-box evasion attack. Typically, this is then formalised as an optimisation problem in terms of the underlying model, which can be solved via a stochastic optimiser like *Adam* (Kingma & Ba, 2014). To ensure that these alterations are subtle enough to be unnoticeable to a human operator, the constraint to be minimised is some distance metric in $\ell_{\{0,1,2,\ldots,\infty\}}$[31] between the altered data and its original. For instance, in a pixel image a bounded ℓ_0 limits the number of pixels that may be changed, while ℓ_2 limits the overall strength of noise added. These adversarial examples typically occur very close to the decision hyperplane; applying too much noise can either accidentally 'push' the data into a classification the attacker did not desire, or it may become humanly perceptible. Since their inception (Szegedy *et al.*, 2013), they have been shown to generalise between models and input vectors (Goodfellow *et al.*, 2014). In the image domain, they have been made transform-resilient (Kurakin *et al.*, 2016), to transfer to textural information on 3D-printed objects (Athalye *et al.*, 2017), and to persist through projection onto surfaces (Gnanasambandam *et al.*, 2021).

[31] These specific metrics are the *Hamming* metric ℓ_0 (number of altered elements in $\mathbf{x}$), *Manhattan* metric ℓ_1, *Euclidean* metric ℓ_2, and the *Chebyshev* metric ℓ_∞ (the largest change to any element).

A more recent formalisation and strengthening of attacks based on raw input data was recently presented by Carlini and Wagner (2017). Around the time of publication, distillation (Papernot, McDaniel, Wu *et al.*, 2016) was proposed as a form of hardening for neural networks expected to perform in adversarial settings where evasion attacks might be common. This work reveals that existing approaches for *generating* adversarial examples weren't strong enough and, accordingly, approaches like defensive distillation are shown to be ineffective. Some future works refer to the methods they propose as CW-$\ell_{\{0,2,\infty\}}$ attacks. Their attacks exceed existing work based on these three well-understood metrics by a more in-depth analysis of the construction of cost functions, a reworked box constraint built around $\tanh(\cdot)$ (as in HDR image tone mapping), and a more nuanced treatment of the effects of discretisation error. By introducing a *confidence factor* κ, they are able to explicitly design attacks which are *transferable* between one classifier and its distilled form, or a network derived from the original by black

box inference.

In practice, inputs to ML classifiers are often heavily pre-processed or undergo some statistical transformation; either to achieve a fixed-size and compact representation or to increase accuracy. In this sense we can refer to the 'true' inputs as belonging to the *problem space*, while the transformed input given to the DNN/ML classifier belongs to the *feature space*.[32] A malware detector would not, for instance, take an executable's binary as its input, and would instead process behavioural features extracted by static and dynamic analysis tools. Of course, the transforms to determine these features are non-invertible and often non-differentiable. Perturbed inputs also need to maintain functionality (in, e.g., malware), and when combined with input validity checks this makes it difficult to create adversarial examples. A recent frontier on enabling such attacks is a framework for expressing input validity and transformation constraints (Pierazzi *et al.*, 2020); if feature transformations are approximately differentiable then they may be used directly, otherwise falling back on genetic algorithms and Monte Carlo tree search.

[32] Image and audio processing are something of an exception to this, where the feature space *is* the problem space. As a result, most adversarial ML research targets these domains for simplicity.

DRL algorithms are equally vulnerable to this class of attacks, despite the fact that their stochastic nature greatly influences the trajectories gathered during training. The meaning of an attacker manipulating the environment is, again, problem space dependent, and most work focusses to some extent on reducing agent performance rather than invoking specific actions. S. H. Huang *et al.* (2017) have shown that this vulnerability to adversarial inputs extends between RL algorithms in white-box settings, while perturbations acquired in a black-box setting on the same NN architecture require greater error bounds to invoke the same loss of reward. An alternative strategy is to directly modify the PPO algorithm, training agents to choose actions with the highest likelihood of making another victim agent perform suboptimally (Wu *et al.*, 2021)—i.e., through this adversarial agent's effect on shared environment state via valid actions.

Defences While there is a great deal of churn in what defences will still be considered 'valid' at any time, there are concrete guidelines on the evaluation of defences (Carlini *et al.*, 2019) to aid in their development (particularly as earlier attempts at defence were not always considered as rigorously as they should be). For instance, an earlier proposed (though now defeated) defence was defensive distillation (Papernot, McDaniel, Wu *et al.*, 2016). Ordinarily, distillation (Hinton *et al.*, 2015) takes the softmax probability scores output by a given model, and applies these as the soft labels to be learnt by a more compact NN architecture. For early evasion attacks, it sufficed to perform this without changing the target model's structure (selecting a higher softmax temperature) such that smoother decision boundaries would be learnt, though this was invalidated by the above CW attacks.

Adversarial training methods have borne some degree of interest. These approaches integrate generated evasion attacks into the training set in some

way; either by explicitly adding generated adversarial examples into the dataset alongside their true labels (Ilyas *et al.*, 2017), or by the direct inclusion of the attack generation method in the loss function (Madry *et al.*, 2018). However, they exhibit some vulnerability to universal black-box attacks and as such should not be used as the sole defence (Tramèr *et al.*, 2018).

In ensemble classification, if many of the individual classifiers disagree then this can represent a high degree of uncertainty about the observed data. Smutz and Stavrou (2016) realise that this can act as a powerful indicator of an evasion attack in progress, and propose *mutual agreement analysis* as a defence on top of the PDFrate (PDF malware) and Drebin (Android executables) malware detection systems. Both of these platforms had well-established adversarial attacks (Maiorca *et al.*, 2013; Srndic & Laskov, 2014), built around the constraint that core exploit functionality must be preserved (i.e., problem-space constraints). When an insufficient amount of the constituent classifiers return the same result, the result returned is that the sample is 'uncertain'—suggesting either a new class of data or evidence of attempted evasion. The approach naturally suits ensemble methods such as *random forests*, but an extension to SVMs is proposed. Moreover, the addition of the 'uncertain' classification acts as a useful metric for continuous training and evolution.

Ensembles of *classifications* around one data point, rather than *classifiers* (i.e., without changing the classifier) have also been considered. Adversarial examples typically occur very close to the decision hyperplane; applying too much noise can either accidentally 'push' the data into a classification the attacker did not desire, or it may become humanly perceptible. This principle is exploited by Cao and Gong (2017), who propose ensemble classification of potentially adversarial data by sampling from the local hypercube—*region-based* classification, rather than standard *point-based* classification. This draws from the observation that most of the volume of the surrounding hypercube lies within the true class region, even for adversarial examples, with the size of this noise chosen to minimise accuracy loss. To learn the true class of an example, we must then choose the class which admits the largest volume of overlap with the sample region: we may approximate this by drawing samples uniformly from this hypercube (e.g., 10 000 points), and observing the output histogram of labels. Nowadays this ties in heavily to the concept of certified defence (Raghunathan *et al.*, 2018) and provable robustness (H. Zhang *et al.*, 2020), which effectively guarantee through training that no perturbation with norm $|\boldsymbol{\delta}| \leq \epsilon$ can alter the output label. *PixelDP* (Lécuyer *et al.*, 2019) connects this notion to differential privacy schemes' formalisation, observing that adding analytically-derived randomness *within* the model can provide certified robustness. This noise may be injected at a hidden layer (i.e., the extracted latent representation), or may be applied directly to the input by training an auto-encoder to generate a noise vector which would map to bounded-strength noise at the target depth. Output classifications are then the expectation of NN softmax outputs computed

over around 300 draws (42 × runtime overhead), with some additional fine-tuning during training to account for noisier inputs. As noise strength increases, overall robustness improves while clean performance degrades.

Recent work suggests that neuron coverage (the pattern of neuron activations throughout an NN triggered by an input) exhibits significant differences between benign and adversarial inputs (Sperl *et al.*, 2020). To perform evasion attack detection, they train a model as normal and generate a perturbation from each input to every other label, measuring the neuron activations for every standard and adversarial input. They then train a fully-connected network to detect adversarial patterns of activation. This works reasonably well, though in a white-box setting this is still attackable at the cost of more visible input noise.

Fully convolutional neural networks, particularly as applied to images, are vulnerable to adversarial 'patches' applied to sub-sections or regions of the input vector. Robust masking (Xiang *et al.*, 2021) tackles this by reducing the size of CNN receptive fields (via ensemble methods or smaller convolution kernels). This forces adversarially triggered features to contribute larger activations than in the undefended case, making their presence detectable. However, this adds a non-negligible clean accuracy cost and adds 10–50 % execution time (large–small NNs).

Tramèr *et al.* (2020) suggest an inherent balancing act between sensitivity and invariance-based attacks—in that defence against one creates a vulnerability against the other. Sensitivity attacks are what we usually consider in this family (a small change which doesn't impact the input's true label), while invariance attacks use a change which *would* alter the true label but is performed in such a way that the model still outputs the old label. The defence in question would be against attacks within some ℓ_p norm ball (i.e., similar pixel/state similarity)—with the findings suggesting that a 'robust' neural network is even more sensitive than an undefended one.

3.6.2 Poisoning attacks

Poisoning attacks are undertaken by an attacker who wants to permanently alter the behaviour of a system which is still training in some way. The key intuition is that an attacker wishes to either choose data points used in training or modify gradient and parameter vectors to affect the model's decision boundaries in some way. For instance, they might aim to subtly adjust model parameters such that a single (chosen) malware sample escapes detection, or to effectively create a DoS by reducing the function of decision-making RL agents. It is crucial to note that this does *not* require white-box access—control over a handful of input samples may be sufficient, as are any collaborative (e.g., FL) or online-learning systems like RL or active learning (Settles, 2010). Semi-supervised approaches are also vulnerable due to their inclusion of a large unchecked and unlabelled dataset (Carlini, 2021).

In this case, an attacker can construct a path in feature space from a labelled point towards a target value (comprising at most 0.1–1 % of the dataset).

Clean-label poison attacks (Aghakhani *et al.*, 2020) combine the insights of DNN adversarial examples to force a classifier to misbehave on a target instance, while all training data appears correct to a human observer. Perturbed data points are selected, forming a convex hull around the target point in feature space. This causes the target to be mapped to the same class as the hull, though it can take around 3 h to launch a successful attack.

Trojan and *backdoor* attacks are a stronger and recent variant of model poisoning (X. Chen *et al.*, 2017). They differ from their precursors in that an attacker aims to keep the victim model's performance unchanged for all typical inputs, while an input of their choice maps to their desired output. An attacker may also have a model learn a 'trigger' in the input (i.e., a relative sequence of state values, or a pair of glasses in an image) which immediately produces their desired label if it is present. These attacks are possible in black-box scenarios with little effort: consider that X. Chen *et al.* are able to achieve both classes of attack in 5–50 injected training samples (versus a training set of size 600 000) without awareness of either the model or training data. White-box scenarios can make these learnt triggers less obvious to an analyst, by including additional terms in the loss function to penalise cases where their feature activations are easy to discriminate from benign inputs (Tan & Shokri, 2020). Concrete attacks have also been proposed, which target FL (Bagdasaryan *et al.*, 2020) and GNNs (Xi *et al.*, 2021).

Defences Earlier work on centroid-distance anomaly detection (Kloft & Laskov, 2010) indicated that online learning systems which assume stationary normality require an exponential amount of poison samples with respect to how far the mean must move. If non-stationarity is modelled via a bounded memory of points then an attacker requires only a linear amount of samples if they control a sufficient fraction of the network throughput. However, this analysis has limited applicability to modern function approximation which encodes far more complex decision surface behaviour, particularly when bounded memories are not kept.

Auror (Shen *et al.*, 2016) attempts to prevent model poisoning in collaborative model training scenarios, and relies upon the observation that gradient updates submitted by users tend to have a known distribution. By performing 2-means cluster detection on *indicative features*, users whose updates consistently fall outside of the benign distribution may be detected and blacklisted.

B. Wang *et al.* (2019) present a pipeline for detecting, identifying and mitigating backdoors in pre-trained DNN models. They observe that the existence of a trigger (mapping into a target class t) makes it easier to adversarially perturb *all other classes* toward t as compared to any other target label. Since

the trigger is shared, it may be reverse-engineered by clustering over all derived perturbation vectors, and thus removed by either model unlearning techniques or input preprocessing. *Februus* (Doan *et al.*, 2020) explicitly pre-processes NN inputs, but observes that past work is limited in accuracy loss when removing trojan patches. In the image domain, the authors apply work on CNN interpretability to identify which image regions are responsible for the output class—with trojan regions domineering compared to the rest of the image content. These pixels are then removed and replaced using an image inpainting algorithm. Backdoors may also be detected by analysing the NN parameters directly. X. Xu *et al.* (2021) train a mixture of smaller 'shadow' models using the same architecture—both clean and trojaned—and train a binary classifier to make this distinction in a white-box setting. In black-box scenarios, they explore the classification of input-output pairs selected to maximise model information, rather than parameter vectors.

3.6.3 Data extraction and privacy

A key consideration of many ML models is that their dense parameter sets encode an accurate, specialised, and *monetarily expensive* logical transform—even if it can't be directly humanly interpreted. This expense arises through the processes required to collect and label input datasets, as well as the vast compute cost associated with training and hyperparameter optimisation in energy and hardware procurement. As a result, learnt models themselves have high monetary value as intellectual property. *Model extraction attacks* allow an attacker to either directly steal knowledge of the architecture and parameter set, or to train a functionally equivalent model using input and output pairs (analogous to distillation). An attacker typically aims to derive a new model with similar or greater accuracy on the same workloads at minimal effort. Furthermore, they make for an excellent precursor to the above evasion and poisoning attacks as a means to elevate a black-box adversary to a white-box one—in such scenarios it is preferable to pursue model fidelity (i.e., matching correct decisions and mistakes alike).

In general, these attacks are launched through acquiring soft labels (i.e., class probabilities) for a representative set of input points from a victim model (Tramèr *et al.*, 2016), and are also applicable to more models than NNs. The presence of soft labels essentially provides strong knowledge about the class and decision boundaries which have been learnt by the victim model, simplifying the training of a new cloned model. Jagielski *et al.* (2020) examine the extraction of high-fidelity models in greater depth, finding that this distillation-like approach is somewhat limited by non-determinism in the SGD procedure and random initialisation of $\boldsymbol{\theta}$ (i.e., peak 93 % fidelity). Their analytic extraction of truly functionally equivalent models based on ReLU behaviour is limited to 2-layer models at this stage. Exact model theft is most possible when directly monitoring PCIe bus traffic, i.e., as performed by a compromised or outright adversarial cloud host provider. Even when

running application code on TEEs to prevent external inspection, access to GPUs or TPUs still requires communication over the insecure PCIe bus.[33] Although additional PCIe traffic, re-ordering, and batching complicate this task, by using locally known models as a reference point it is possible to extract the sent compute kernel in spite of proprietary driver behaviour.

[33] While confidential models could in theory be run on the CPU using TEEs, most capable Intel CPUs have an enclave memory limit of around 128 MiB. More recent offerings are beginning to extend this into ≥8 GiB (Menon, 2021), though this still limits the achievable throughput one can attain versus a dedicated accelerator.

The question of which input data should be used to launch such an attack is also worth consideration; extracting a model in fewer samples means that not only can a model be cloned in less time, but in fewer discrete interactions with the victim (lowering the chance of detection). Assuming a similar (labelled) training set to the target, Y. He *et al.* (2021) use mutual information analysis to aid in this process. They reduce the input dataset to learn what the most valuable points in their own dataset to query from a victim's model are. If such labels are not known, semi-supervised learning techniques can aid in choosing a pared down set of queries (Jagielski *et al.*, 2020). Construction of a viable query dataset with no knowledge of the victim model's training distribution is also possible (Truong *et al.*, 2021)—generative models may be used to create input samples which maximise disagreement between the clone and victim models.

The opacity of ML models has raised the question of how much information they implicitly contain about the training data they are based upon, i.e., whether training data may be extracted or inferred. *Membership inference* attacks take a given model and input $\mathbf{x}$, and ask whether $\mathbf{x}$ was used in the training of this model—one of the main risks being that sensitive data may be inferred using a speculative query. Models can be expected to have higher confidence about samples they were trained upon; this can be exploited to train a classifier on input/soft label pairs using smaller 'shadow' classifiers (Shokri *et al.*, 2017). While this was originally thought to arise from overfitting, analysis of language models such as GPT-2 suggests this is not the case (Carlini *et al.*, 2020). This same analysis is able to propose and generate prompts to extract such memorized data (including personal data), however accuracy of generated 'members' is limited to around 33.5 %. Collaborative learning settings are vulnerable to membership inference on a per-participant level from their gradient updates (Melis *et al.*, 2019), allowing properties of subsets of classes to be detected according to learners' (non-*independent and identically distributed* (IID)) training data. *Model inversion* attacks focus on extracting information about an input or the space of inputs which would map to a given output class (Fredrikson *et al.*, 2015), e.g., extracting the face of an individual from a facial recognition classifier. These attacks present a real threat in concert with model extraction, for instance Tramèr *et al.* (2016) are able to extract images of faces which are "visually indistinguishable" from the real training data of a facial recognition classifier. *Attribute inference* attacks, where an attacker has most of the information of a member and seeks to derive the remainder, requires a stronger adversary who can tell apart true inferred members from nearby data points (B. Z. H. Zhao *et al.*, 2021).

Defences Given that model extraction attacks are enabled by information sharing, the simplest defences involve limiting the amount of output information given by a classifier. For instance, rounding confidence values and soft labels or outright removing many output probabilities can offer some degree of defence (Chandrasekaran *et al.*, 2020; Tramèr *et al.*, 2016). Sampling a model's parameters (potentially as part of an ensemble) may also serve to make the decision boundary harder to learn (Chandrasekaran *et al.*, 2020). Attacks designed to maximise query information exhibit characteristic patterns, differing in aggregate information per query versus normal users. This principle has been applied to detect extraction attacks by monitoring the distribution of information content of input-output pairs (Juuti *et al.*, 2019; Kesarwani *et al.*, 2018).

Differential privacy (Abadi *et al.*, 2016) is a powerful formalism for bounding the amount by which any individual training datum can affect the overall behaviour of the model. Typically, this is achieved by injecting noise with magnitude below an analytically derived bound to the objective/loss, gradients, or output vectors. By construction, this prevents model and attribute inference attacks, while limiting the amount of information which can be gleaned in collaborative learning settings. However, a more secure privacy budget is directly at odds with final model accuracy, and relaxations to the differential privacy formalism invite additional vulnerability to model inference attacks (Jayaraman & Evans, 2019). In addition, recent work on model unlearning (by dataset sharding and slicing) can be used to truly remove training members from the model by partial retraining (Bourtoule *et al.*, 2021).

3.7 Summary

I have introduced and discussed a wide variety of problems in the networking domain where ML and RL techniques are able to provide tangible improvements in performance and resilience of modern networks. We have also covered the basics of ML techniques and function approximations, as well as the intuition and MDP formalism underpinning RL algorithms. Finally, we have examined modern RL algorithms, concessions and design decisions required to bring RL and ML to network devices, and open challenges pertinent to DDN.

Unfortunately, the security context around DDN remains an uncomfortably open question. While not all the attacks presented can *immediately* generalise to networks or other problem-space representations, the absence of research into specific attacks in this field does not inspire confidence. The interaction model of networks *does* present a useful level of indirection, such that model extraction and similar approaches are harder to perform. For instance, while exact traffic routing or queueing decisions should enjoy some degree of isolation from the actual effects they cause on a monitored flow—

there are many potential hypotheses behind a decrease in maximum flow performance. Efficacy aside, this casts significant doubt on whether we can (safely) elevate RL or other online approaches from an interesting research question towards real-world deployment.

Chapter 4

DDoS Prevention by Multi-agent Reinforcement Learning

Network anomaly detection and intrusion detection/prevention are continually evolving problems, compounded by the partial, non-IID view of data at each point in the network. Looking ahead to our discussion of DDoS attacks in section 4.1, attacks and anomalous behaviours evolve, becoming more sophisticated or employing new vectors to harm a network or system's confidentiality, integrity, and availability without being detected (Bhuyan *et al.*, 2014). These attacks and anomalies have measurable consequences and symptoms which allow a skilled analyst to infer new signatures for detection by signature-based classifiers, but unseen attacks may only be defended against after the fact.

We've already discussed how ML and DDN were hoped to make this domain easier to solve—e.g., automatic detection of attack signatures, reliable anomaly detection—and the operational limits which have become clear in section 3.1.3. Consider on one point in particular, namely the availability of useful training data. In many of these cases, anomalous events still require human expertise to label and detect; to complicate matters, this effort must be sustained in the face of network evolution. For certain classes of problem, e.g., volumetric DDoS attacks, system health corresponds to wider load and performance metrics which are typically more easily known. It is here that RL offers another perspective. Recall from chapter 3 that RL agents operate by following a *policy* to interact with or control a system, while at the same time using observed performance metrics and deliberate exploration to dynamically improve this policy. In this way the role of an RL agent differs from that of a standard classifier, adaptively reacting to threats by assuming the role of a feedback loop for network optimisation, typically to safeguard service guarantees. In a sense, this allows us to "overcome" some of the difficulties of the detection problem by monitoring *performance characteristics and consequences* in real-time; by looking for (and controlling) the effect rather than the cause.

What RL approaches do exist are aimed towards the task of adaptive online DDoS mitigation, and rely upon learning to control probabilistic packet drop. These have concrete weaknesses compared to the reality of network traffic. Section 3.1.3 presents my analysis of the existing work for this task—*Marl* (Malialis & Kudenko, 2015)—particularly how it fails to account for congestion-aware traffic (i.e., TCP) and environments with high host density per egress point such as ISPs or datacentre networks. As a result, they achieve poor protection of legitimate traffic due to an overly coarse view of the network and the dominance of congestion-aware traffic in today's networks (section 4.1.2 and appendix A). However, there are limits to how we should infer these properties given network evolution—we must remain protocol- and content-agnostic to offer future-proofing against the rollout of protocols like QUIC.

This chapter considers how online RL can be used to defend networks from volumetric DDoS attacks, agnostic of the protocols of carried traffic, and is based upon 'Per-Host DDoS Mitigation by Direct-Control Reinforcement Learning' (K. A. Simpson, Rogers & Pezaros, 2020). I first explain the existing threat and defence landscape of such attacks (section 4.1), then reiterate the motivation for RL as a solution (section 4.2), before defining the threat model for attackers with respect to known DDoS attack methods and the security context of ML (section 4.3). Section 4.4 then outlines the design and rationale behind new agent designs built to improve on the failings of past RL works, by making decisions on a per-flow or per-source basis. This includes algorithmic modifications to learn from and control many traces simultaneously, achieve faster convergence by per-tile updates, and to better learn from individual features. I describe a feature space built on a mixture of global and local state, reward functions tailored to different attack classes, and contribute two action models and their risks (*Instant* and *Guarded*). The *Guarded* model is inspired by past work on algorithmic DDoS prevention, as an example of how the integration of domain-specific knowledge can lead to more effective RL agents in shorter timescales. Section 4.5 then describes how state measurement and installation of actions could be managed in an SDN deployment. To determine *which* per-flow features are worth using for DDoS control and mitigation, I then present qualitative and quantitative analysis of a large selection of these metrics for different agent designs on varied protected traffic types (section 4.6). Section 4.7 then details my implementation of reactive simulations of HTTP and VoIP web-server traffic, designed to test system characteristics that packet trace playback fails to capture. Sections 4.8 and 4.9 then describe and show empirical performance measurements of the two new agent designs against existing RL DDoS techniques, and algorithmic works via *SPIFFY* (Kang *et al.*, 2016b), reuniting two divergent strands of research and grounding the study of RL-based DDoS defences. I conclude with some discussion on the significance of the results and wider security implications of this solution in particular (section 4.10), and summarise in section 4.11.

4.1 Distributed denial of service

While computer networks are prone to all manner of operational problems on account of their gradual, continued construction via many complex interlocking systems, we train our focus here on *Denial of Service* (DoS) and *Distributed Denial of Service* (DDoS) attacks.[1] DDoS attacks are concentrated efforts by many hosts to reduce the availability of a service, typically to inflict financial harm or as an act of vandalism. Attackers achieve this by either exploiting peculiarities of OS or application behaviour in *semantic attacks* (e.g., *SYN flooding attacks*), or overwhelming their target through sheer volume of requests or inbound packets (*volume-based attacks*) (Jonker *et al.*, 2017). Hosts often participate unwillingly, typically having been recruited into a *botnet* by malware infection to be orchestrated from elsewhere (Antonakakis *et al.*, 2017).

[1] A vast array of other, keenly relevant problems are briefly explained while motivating the DDN use cases presented throughout section 3.1.

Why focus on this problem in particular? The primary reason is that its scale and impact presents a constant threat to any Internet service. Exhausting all of a server's resources (or those of the infrastructure providing a path to it) ensures that it cannot be accessed—causing financial losses, silencing information sources, or creating downstream service breakages. Some services, such as those associated with game hosting, are likely to be targeted simply for competitive advantage or reputation (Pinho, 2021). Accordingly, DDoS attacks are often nicknamed an '800-pound gorilla' (Czyz *et al.*, 2014) on the wider Internet. Their reach is, however, made all that much greater by the centralisation of many websites and servers to cloud and hypergiant infrastructure. Consider volumetric attacks on Dyn (1.2 Tbit/s), who at that time hosted key resources for Twitter, Spotify, and Netflix (Hilton, 2016), the web host OVH (1 Tbit/s) (Klaba, 2016), and the Github code hosting platform (1.35 Tbit/s) (Kottler, 2018). Amazon's own services have been an attractive target on several occasions: S3's *Domain Name System* (DNS) servers were hit by an attack of unknown size in October 2019 which was unmitigated (McCarthy, 2019), while AWS successfully resisted 2.3 Tbit/s of traffic mere months later (Lyon, 2020). Even individuals' blogs such as KrebsOnSecurity (665 Gbit/s) (Krebs, 2016) have been high-profile targets. The more solutions and insight the research community can provide, the better.

The second reason is that DDoS defence scenarios may be a representative example of the kinds of closed loop control that DDN is exceptionally well-suited to. Target servers and infrastructure expose useful state such as link utilisations, queue depths, and service metrics; an influx of attack traffic has noticeable effects on this state, and taking the 'right' control plane actions (e.g., blackholing specific traffic sources or protocols, filtering out attack packets) should move the network's state closer to some degree of normality. At the same time, DDoS strategies evolve over the course of a single attack (Kang *et al.*, 2016a), potentially leading to a nuanced (and difficult) measure→infer→act loop. An ideal, human-designed solution to this con-

trol loop is made tricky by the complex interplay of attack dynamics with many existing elements of the network: protocol distribution and behaviour, application behaviour, and the gradual evolution of benign traffic. For this reason, I focus on DDoS attacks as a particular use case in this thesis—in turn, chapter 4 is dedicated to improving automated, data-driven means for their solution.

Jonker *et al.* (2017) offer an in-depth analysis and taxonomy of the landscape of DoS attacks. They observe that Denial-of-Service is most commonly achieved through *resource exhaustion*—either at the target server or the infrastructure serving it. Attacks may then be classified on two orthogonal axes: *Direct vs. Reflection* and *Volumetric vs. Semantic*.

Direct Attackers send packets directly towards their target. Random IP spoofing is often used to make blocklisting more difficult, but leaves evidence of an attack and its characteristics due to *backscatter*, visible to network telescopes (D. Moore, 2003; Richter & Berger, 2019).

Reflection Attackers send traffic to a *reflector*, spoofing the source IP of packets to match that of the target. The reflector sends replies to the target, often *amplifying* them in the process.

Volumetric DoS is achieved by *resource exhaustion*—CPU or RAM usage at a target host, or occupying and overflowing transmission buffers along key traffic routes. These can be service agnostic, but in some cases rely on buggy behaviour of other software as their main mechanism.

Semantic DoS is achieved by *exploiting program logic*, for instance to crash a target application server. These are often tailor-made for a particular service or its deployment environment, such as *teardrop attacks* against a host's TCP stack.

We'll focus mainly on volumetric attacks (direct and reflection), as these are the attack vectors applied in the listed, real-world attacks.

4.1.1 Volumetric attacks

Amplification attacks Amplification attacks abuse network services with small request bodies and large responses, causing a typically benign service to forward traffic on an attacker's behalf by *reflection*—spoofing the source IP of requests to that of the intended victim. An attacker requires that their AS doesn't prevent IP spoofing at egress. In exchange, they are able to split their upstream bandwidth across many reflectors to gain higher volumes of attack traffic from multiple sources without revealing their own IP to the victim. UDP-based protocols are the typical basis for such attacks, as the transport is connectionless.

While DNS is the most well-known vector for amplification, Rossow (2014) presents an in-depth survey of a wide variety of other vulnerable protocols alongside a rough census of abusable servers. He examines network services (SNMPv2, *Network Time Protocol* (NTP), DNS, NetBIOS, SSDP), legacy services (CharGen, QoTD), peer-to-peer networks (BitTorrent, Kad), online games (Steam, Quake 3) and externally abusable botnets (ZAv2, Sality, Gameover). Scanning for 10^5 amplifiers of a popular service can be done in minutes, making NTP particularly dangerous due to its prevalence and high amplification rate. Furthermore, he notes that DNSSEC can exacerbate the problem by its addition of large signatures to message payloads.

Kührer *et al.* (2014) build further upon this census; they find significant overlap between servers who expose different vulnerable services, connect these services to OS fingerprints, and use DNS proxies to enumerate the ASes who allow IP spoofing. They find that many eligible reflectors tend to lie behind dynamic IP addresses and so undergo significant churn (meaning an attacker must often re-scan every few days/weeks). This is not the case for certain protocols like NTP, where the server list remains far more stable. The authors also explore the amplification potential of all TCP-based services—given that well-known protocols like HTTP cannot be blocked in most infrastructures, an attacker can abuse retransmissions of the handshake (`SYN-ACK`) to achieve an amplification factor up to 20 × if the receiver doesn't send `RST` responses.

NTP quickly became the attack vector of choice, according to Czyz *et al.* (2014). They find that most vulnerable amplifiers are *end-hosts*, typically offering 4 × amplification. At the time of publication, they observed that NTP amplification attacks had risen in volume by ~1000 ×, though were slowly declining; 85 % of attacks over 100 Gbit/s relied upon NTP reflection. The decrease, they posit, stems mostly from vulnerable servers being patched in response to recent bulletins making the risk clear to server operators. They observe that, after the patch period, many of the remaining vulnerable servers are sparsely distributed (rather, the patched servers are clustered under IP blocks). This is in line with the (un)cleanliness hypothesis put forth by Collins *et al.* (2007). Of greatest concern was the presence of 'mega-amplifiers' offering 10^3–10^9 × amplification due to the presence of network loops.

Kührer *et al.* (2015) investigate the landscape of *open recursive DNS resolvers*, one of the major enabling factors for DNS amplification attacks. Many of these DNS servers run old and vulnerable software, and are very highly represented (67 %) by consumer routers linked to dynamic IPs.

As of 2017, the distribution of amplification attacks over UDP protocols was observed to roughly have the pattern NTP>DNS>CharGen. This is in spite of the evidence put forth by Czyz *et al.* (2014), which suggested a decline of NTP-based amplification attacks.

It must be reiterated that new amplification DDoS vectors arise due to software bugs and misconfigurations even today. *TsuNAME* (Moura *et al.*, 2021) is a recent example, where the presence of recursive DNS dependencies causes traffic amplification toward authoritative name servers. While this cannot be directed to an arbitrary target *per se*, this presents another vulnerability in critical infrastructure that administrators must be aware of.

Link-flooding attacks *Link-Flooding Attacks* (LFAs) or *transit-link* attacks are another volumetric DDoS vector which has come to light (Kang *et al.*, 2013; Studer & Perrig, 2009). In contrast with typical direct and reflection-based attacks, malicious actors here do not forward traffic directly to their intended victim. Instead, they use their bandwidth to communicate with as many legitimate or dummy servers as they can such that the outbound traffic of all attacking clients aggregates at a common point in the Internet. This exhausts the resources of a target AS or set of bottleneck links needed to reach their intended victim, and traffic appears for all intents and purposes as a completely uncorrelated set of source and destination pairs. Since the traffic only ever aggregates in, e.g., ISP networks, target servers never see any attack traffic themselves. The need for many source nodes means that attackers practically require botnets for LFAs to be feasible (Smith & Schuchard, 2018); however, Internet-of-Things devices and other insecure machines are often recruited for this purpose via malware like *Mirai* (Antonakakis *et al.*, 2017).

4.1.2 Contributing factors in the detection problem

Variation in normality Benign traffic is in no way 'normal', and is often composed of a variety of heterogeneous traffic classes acting in different ways. Protocol families respond differently to both administrator actions *and* the presence of attack traffic; mainly, this difference is seen between congestion-aware and -unaware flows. At a high level, congestion-aware traffic tends to scale its rate up to its maximum fair share and scales down in response to congestion signals such as packet losses (e.g., TCP), while congestion unaware traffic ignores these requirements (e.g., most UDP flows).[2] Consider probabilistic packet drop at a rate $p \in (0, 1]$—pushback (Mahajan *et al.*, 2002). Loss-ignoring and CBR traffic's send rate will scale in proportion to $1 - p$. TCP Reno and the like exhibit greater falloff proportional to $1/\sqrt{p}$ courtesy of the Mathis equation (Mathis *et al.*, 1997), with a kinder $1/p^{0.75}$ for TCP Cubic (Rhee *et al.*, 2018), inflicting greater collateral damage than expected on misclassified but legitimate flows. Even then, congestion-aware traffic's precise response depends on CCA design, protocol implementation details, and the nature of the carried traffic (e.g., bulk transfer vs. RPCs).

Attack traffic may well share a feature with a non-dominant family of protocols, at which point basing a defence on that mechanism will result in

[2] This distinction is not quite as simple as 'TCP & UDP'. Due to middlebox-driven Internet ossification, *QUIC* (Langley *et al.*, 2017) and *SCTP* (Stewart, 2007) are carried over UDP. Respectively, they are and can be congestion-aware. BitTorrent's *µTP* (Rossi *et al.*, 2010) builds on UDP to offer a lower-latency congestion-aware transport. Finally, adversarial replayed TCP traffic (e.g., `SYN` floods) is of course congestion-unaware.

harming or blocking that legitimate traffic—*collateral damage*. For instance, CBR traffic such as VoIP flows are unlikely to respond in a meaningful way to a change in their bandwidth allocation, short of recording and reporting packet loss statistics. In contrast, most congestion-aware flows (including LFA sources) will respond to bandwidth expansion and contraction, with LFAs having little to no response compared to legitimate traffic (Kang *et al.*, 2016b).

Finally, the exact proportions of this heterogeneous mixture vary over time and between AS classes. Consider a point estimate of sorts obtained by analysing the 2018 CAIDA traces (CAIDA, 2018), shown in appendix A. On this tap of ISP traffic, congestion-aware traffic makes up at least 73–82 % of packets; varying over time and the link's direction. The corollary is that congestion-unaware traffic makes up at most 27–18 %—a significant fraction of collateral damage, if we are careless around our defence and detection model in the above example. The first figure includes some peak 3.26 % of QUIC traffic in the *Sao Paulo to New York* direction. Variability extends also to the behaviour of flows *within* a protocol. This presents in some cases as long-tailed distribution between more numerous, shorter *mice* flows and longer *elephant* flows (Pan *et al.*, 2003). A consequence is that punishing actions can have a greater relative impact on some flow classes over others (in this case, packet losses would have the greatest impact on mice FCTs).

Evolution Just as new attacks and attack vectors arise over time, so too does the rest of the network evolve. New protocols such as *QUIC* (Langley *et al.*, 2017) come into use in the Internet at large, and can achieve near-instantaneous widespread adoption via the backing of hypergiant network operators. New CCAs such as *BBR* (Cardwell *et al.*, 2016) are deployed to improve flow bandwidth utilisation, but lead to observable changes in flow-level behaviour. At the aggregate level, heavy hitter flows have seen noteworthy increases in duration and rates over a 13-year time horizon, as has the mice-elephant balance (Bauer *et al.*, 2021). Detection and mitigation solutions must be aware of these eventualities to protect legitimate traffic over longer timescales.

4.1.3 Defences

According to Jonker *et al.* (2017), the most-used techniques in deployment are *DDoS Protection Services*. While typically proprietary in nature, we see a split between *cloud services*, *in-line systems* (i.e., middleboxes) and hybrids thereof. Cloud services (traffic scrubbers) are known to be most appropriate for handling volumetric attacks and are externally hosted, analysing and filtering out malicious traffic by having services redirect all inbound communication for processing. The act of redirection is often made cheaper and feasible by the use of selective BGP advertisement or DNS modifica-

tion, aided by reverse proxy or *generic routing encapsulation*. Amongst these, BGP-based diversion is most effective where many hosts must be protected, and DNS works most reliably for single-host installations. In-line systems, hosted within a service's domain of control, are most useful for handling semantic attacks as these often admit *attack signatures* (since they must exploit a particular bug in the server). Similarly, such attacks tend not to exhibit long-term characteristics that cloud scrubbers might use to aid detection, as many of these attacks present themselves as a single packet.

DDN solutions to DDoS attacks have been examined through the literature, such as Braga *et al.* (2010), *Marl* (Malialis & Kudenko, 2013; 2015), and *Athena* (Lee *et al.*, 2017). Section 3.1.3 explains these, alongside their drawbacks and experimental shortcomings, in detail. Marl in particular has design flaws which are placed under great scrutiny, motivating the improved RL work I develop in the remainder of this chapter.

Rossow (2014)'s suggestions are mostly prophylactic. At the AS-level, IP spoofing by internal clients must be prevented. Protocol designs should be hardened with session handling *à la* QUIC or Datagram TLS at the expense of latency, enforcing greater symmetry of request and response sizes, and rate limiting the frequency of per-client responses.

Honeypots such as *AmpPot* (Krämer *et al.*, 2015) can play a key role in the detection and mitigation of volumetric attacks. Fake amplifier services hosted by legitimate authorities, which appear to be useful amplifier nodes to malicious actors, may be included in the set of reflector nodes when attacks are launched. As a result, infrastructure providers can receive early notification of attack targets and the protocols which must be blackholed.

SPIFFY (Kang *et al.*, 2016b) aims to remedy LFAs by observing how flows from each source respond to a sudden increase in available bandwidth. Kang *et al.* realise that bots participating in an attack are often unable to match this bandwidth expansion due to having already saturated the capacity of their outbound links, while legitimate flows typically speed up to match the new fair-share rate. Due to the class of attacks it is designed to defend against, SPIFFY is intended to be deployed within ISP networks. However, they find that computing per-flow routes to offer this expansion is expensive on real networks (14 s in the Cogent topology), and achieve only low expansion factors which require more rounds of filtering. Finally, by definition their assumptions cannot extend to CBR traffic (e.g., UDP VoIP traffic), which as we know from section 4.1.2 and appendix A makes up a sizeable proportion of network traffic. Only congestion-aware traffic will correctly alter its behaviour under this action and response model.

Smith and Schuchard (2018) present techniques based on AS-level routing to tackle both transit-link and flooding-based attacks. This view is taken due to the perceived cost of per-stream classification and inherent sensitivity to adversarial examples or crafted input. The approach is creative, relying

upon BGP *fraudulent route reverse poisoning* to preserve traffic to a target AS, but unlike SPIFFY the approach doesn't actually *remove* the congestion. Because of this, traditional flooding-based attacks aren't fully alleviated.

SENSS (Ramanathan *et al.*, 2018) aims to help hosts and *endpoint-servers* communicate upstream with ISPs. The rationale is that although DDoS traffic can be filtered at any point along its path, it will impact less of the network if it is filtered *close to its source*—this observation holds true in all attack classes (direct, reflection, LFA), which exhibit a tree-like pattern of traffic. This information currently propagates through human channels, eventually leading to traffic blackholing being performed by key ASes. The core idea is that the *victims* should be given responsibility for intelligence and decision-making, who pass on their requests to ISPs (alongside ample payment). They are able to show that this approach functions for multiple algorithms—including using NAT for true outbound requests as a mechanism for reflector filtering close to the source, similar techniques to others to "route around" the congestion added by LFAs, and location-based filtering for signature-free attacks.

S. Simpson *et al.* (2018) propose the *Antidose* collaborative solution. ASes ask one another to install allow- and block-list filters to represent the interests of their own transit traffic while disallowing known-bad sources and ASes. Hosts and agents must perform proof-of-work challenges attached to flow cookies to become eligible for allowlisting (which is verifiable by any other node)—however, this requires some degree of re-architecting the network stacks of all hosts.

Some collaborative solutions appear to hinge on the condition that HTTP and TCP sessions can be reliably held over the saturation zone between high-priority endpoints. Alternative channels may be possible through elected proxies or UDP-based mechanisms like *DDoS Open Threat Signalling* (DOTS) (Dobbins *et al.*, 2021). DOTS provides an architecture for network operators to enumerate, discover, and communicate with DDoS mitigation services, with who they can exchange telemetry information and explicit mitigation requests.

4.2 Motivation

Moving beyond the overt benefits of choosing RL-based defences for coping with evolving or ongoing control problems, I believe that there are concrete reasons for their use here. We have seen that for other domains in particular, misclassification is a serious problem, which can introduce *collateral damage* in the context of DDoS prevention. In theory, the feedback-loop-like model allows us to monitor flows *after* an action is taken to allow forgiveness of mistakenly punished flows. This does rely upon the ability to take a flow-by-flow view of the state space, but if we can combine knowledge of

current state with the last applied action, then perhaps a flow which falls off identically to a legitimate flow can be rescued.

Other studies suggest that there are particularly useful features which make the task of online DDoS flow identification feasible. Aggregate network load measures observed at various locations suggest the overall health of a network (Malialis & Kudenko, 2015), for instance high link occupations but few successful requests reported by a target server might be an indicative feature. Similarly, the ratio of correspondence between pair flows can suggest asymmetry, and in many cases illegitimacy (Rossow, 2014). Generic volume-based statistics (counts, counts per duration, average packet sizes) have seen effectiveness in such as kNN classifiers trained to detect DDoS attacks in progress (Lee *et al.*, 2017). Most importantly, there is evidence showing behavioural changes in response to bandwidth expansion (Kang *et al.*, 2016b), suggesting similar artefacts might arise after throttling, packet drop, or other interference.

4.3 Threat model

An attacker's goal is to minimise the fair-share bandwidth allocation that a server can give to hosts, and they are expected to act rationally in its pursuit. Threat actors are external and act intentionally, aren't expected to be *Advanced Persistent Threats* (APTs), and likely range from hacktivists to moderately funded adversaries. We assume that attacks will be volumetric DDoS attacks with the structure of an *amplification attack*, and that traffic aggregates at the target (unlike in a transit-link attack or LFA). The addresses of the set of unwitting reflector nodes are visible to the target, though any bots taking part in an attack or the machines those bots control are not revealed to the target without communication with third-party organisations such as upstream ISPs. The discovery of any reflector by some defence system does have a cost to the attacker—there is a particularly large (yet finite) supply of viable reflector nodes (Rossow, 2014), but the constraints that each has a large upstream bandwidth and support for high-amplification protocols narrow this pool.

We do not assume that an attacker has white-box access to an agent's policy, or that they will attempt to intelligently modify flow/system state to indirectly control an agent (Carlini & Wagner, 2017; S. H. Huang *et al.*, 2017; Papernot, McDaniel, Jha *et al.*, 2016; Papernot *et al.*, 2018)—the kinds of evasion attack considered throughout section 3.6. While they may be able to perform some degree of reverse engineering by observing the health of their own legitimate canary flows, "stealing" the policy through observation (Tramèr *et al.*, 2016), this would require an attacker to indirectly observe the effects of (probabilistic) actions applied to their traffic—in addition to effects imposed by other flows competing for resources. Moreover, gaining feedback on the fate of attack packets is less feasible with connectionless

traffic, and doubly so when it is generated by an amplifier not under the adversary's control. Investigating whether perturbations applied to flow state would persist in volatile network traffic statistics falls also outside of the scope of this work. The same observation extends to the possibility of poisoning attacks (Han *et al.*, 2019). These are APT-level capabilities, whose exploration presents a rich source for future work.

4.4 Per-flow RL agent designs

Given existing works, my hypothesis here is that the best method for advancing past the current shortcomings of RL-based DDoS mitigation is to design agents such that filtering decisions are computed per flow. However, these alterations must account for computational constraints imposed by the deployment environment. For instance, the amount of flows passing over an agent is unbounded for larger networks, potentially inflicting huge inference and monitoring costs on agents. These require dedicated, careful handling. I describe and justify my approach, domain-specific algorithmic improvements, and present two action models, one of which draws on domain knowledge introduced by SPIFFY (Kang *et al.*, 2016b).

4.4.1 System design and assumptions

A deployment environment is a network with a set of *ingress/egress points* from its domain of control, through which traffic can flow, and a set of protected *destination nodes*. These nodes may be services, servers, or in the case of ASes and transit networks, egress points leading to other networks. Agents are co-located with each egress switch (i.e., k ingress points from other ASes require k agents), all employ the same action model/design, and control the proportion of upstream packets from each external host to discard. Each destination node s has a maximum capacity on its link utilisation, U_s.

We assume that the deployment environment is a moderately complex SDN-capable network, because the paradigm offers features which can directly benefit RL agents acting within its confines. The OpenFlow protocol allows a controller (or other authorised hosts) to install complex actions, forwarding rules and logic into a switch at runtime. For simplicity at this stage, all agents or learners run on commodity host machines. Furthermore, networks of this kind more naturally enable the use of NFV, allowing relocation and easy installation of learners—e.g., as examined by Jakaria *et al.* (2019). Agents communicate with their co-hosted OpenFlow-enabled switches—running a modified version of OVS ('Open vSwitch', 2018; Pfaff *et al.*, 2015)—to install probabilistic packet-drop rules.

This system design applies to both software-defined and traditional net-

works of arbitrary shape and size. Only the ingress and egress nodes from a network need to be OpenFlow-enabled, as it is advantageous to perform filtering as close to a flow's source as possible.[3] In a traditional network, each agent has exclusive control over its switch's tables. In a fully software-defined network, these agents require exclusive control over the first table, forwarding legitimate packets to subsequent tables managed by the network's controller. The main difference is that a traditional network needs this additional hardware, and does not allow an operator to dynamically determine where the "edge" of their network lies through VNF relocation.

[3] Note that, depending on the size of the target network, this needn't be a hardware OpenFlow switch. Some degree of horizontal scaling could be achieved with OVS instances. Similarly, a P4 dataplane device could fill the same niche, making the 'probabilistic packet drop' primitive similarly easy to integrate.

4.4.2 Algorithm

To make decisions cheaply and at low latency, we use *semi-gradient Sarsa with tile coding* as described in eq. (3.6) and section 3.2.1, rather than using neural networks or more complicated function approximators. Exploration is introduced via ϵ-greedy action selection, linearly annealing ϵ to 0 over time. Each agent has its own internal parameter vector $\boldsymbol{\theta}$, and agents do not share their weight vector updates with one another (but may share experience and traces with one another).

Although the choice of a classical RL method likely brings lower theoretical performance, there are significant reasons to favour such methods; these include lower latency decision-making, lower energy usage, reduced model complexity (and training time), the availability of necessary hardware, and simpler decision boundaries. This aligns with our goal of quick online learning, and faster adaptation to aggregate changes in traffic without introducing dedicated tensor processing hardware to networks. Simpler decision boundaries reduce the risk of overfitting and unexpected behaviour, and we expect that the simplicity of tile-based policy computation will also greatly aid interpretability of anomalous action choices.

When choosing a learning algorithm, I compared against Q-learning as well as methods based on *eligibility traces* such as Watkins's Q(λ) (Sutton & Barto, 2018, pp. 312–314) and Sarsa(λ) (Sutton & Barto, 2018, p. 305). Preliminary experiments found that Sarsa offered the best performance and behaviour.

Action rate I adapt Sarsa to prioritise rapid response to changes in network state and to visit as many state-to-state transitions as possible for effective learning. To this end, we allow agents to make many decisions per timestep. We maintain the last state-action pair associated with each source and destination IP address, and calculate any actions for the flows which still exist. Finally, we update $\boldsymbol{\theta}$ using each available trace and the reward signal from the relevant destination. As exploration still occurs for each action, this approach reduces ϵ multiple notches every timestep. In turn, we increase the

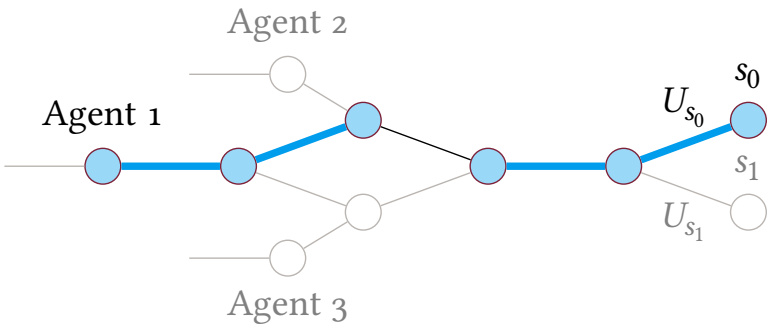


Figure 4.1: Global state selection for a flow between an external host and server s_0 which passes over Agent 1. All nodes in the path taken through the defended network are filled in blue, and all link load measurements which are chosen for action computation are indicated with a thick blue line.

annealing window for ϵ by a factor of 2.67 so as to preserve exploration over time, by accounting for the greater volume of decisions being made.

Per-tile updates While the standard formulation of eq. (3.6) updates the value of all tiles identically (by a scalar $\alpha\delta_t$), I found it more effective in this use case to compute a different temporal difference value *for each tiling*. While we make use of the sum of all tiles' action value estimates when making decisions, each tiling is updated using only its own contribution, allowing us to set α to a higher value without divergence. A crucial observation is that value updates to each tile can move by different values in different directions, converging on effective estimates sooner.

Decision narrowings When learning control on the basis of a tile-coded state space from high-dimensional state, assignment of credit to individual features for each decision is difficult because all tiles have identical gradient. To combat this, with probability ϵ an agent will mark a flow as being governed by a subset of the state space for the next 5 decisions. Each agent chooses actions on that source/destination pair using one element of local state, the global state, and the bias tile—the latter two are included to strike a balance between accuracy and correct credit assignment.

4.4.3 Feature space

The improved state space combines elements of global state, network link load observations as used in past work such as Marl (Malialis & Kudenko, 2015), with per-flow measurements. Each is tile-coded with 8 tilings and 6 tiles per dimension, using the windows described in table 4.1.

Global state is a vector of load values in $\mathbb{R}^4$ (Mbit/s) depending upon the bandwidth measurements regularly received from monitors in the environment. For any flow, an agent first computes the path it would take through

the network. The incoming link utilisations of the first hop, last hop, and tertiles of the path are then tile-coded together, giving a fixed-size representation of network characteristics along the traffic path. In the event that the path from an agent to its destination is shorter than 4 hops, we simply duplicate the load measurement of a middle hop or the last hop (in order of preference). Figure 4.1 illustrates the process.

Global state is built in this way to offer compatibility with multipath, multi-destination networks, offering support for diverse deployments from endpoint servers to transit ASes. Computing the path from agent to destination is not computationally expensive. Multipath routing is often fast since typical ECMP algorithms simply hash a packet's flow key, and are deterministic to provide consistent QoS to hosts.[4]

[4] It's expected that most deterministic load balancing schemes should be trivial for hosts or controller machines to compute given up-to-date topology information. More dynamic schemes which balance adaptively on a flowlet or per-packet model are somewhat out of scope—this might be captured by an expectation in link loads over all valid paths.

Each one of the per-flow features included in the state vector is described and analysed throughout section 4.6. Each feature is tiled separately, with the exception of packet in/out count (per-window and total), mean in/out packet size, and Δ in/out rate, which are combined with the last action taken. Rather than having the network push the data to an agent, the agent requests this information about active flows periodically to isolate it from non-control-plane traffic and to eliminate the risk of resource exhaustion by excessive requests.

4.4.4 Reward function

Every timestep t, each destination node s generates a reward signal $R_{s,t}$. Assume, for now, that each destination has access to some classification function $g(\cdot)$ which estimates the volume of legitimate traffic received, and expects to receive $traffic_s$. Denoting the upstream, downstream, and combined loads as $\text{load}_t^{\uparrow}(s), \text{load}_t^{\downarrow}(s), \text{load}_t^{\updownarrow}(s)$ at this node:

$$c_{s,t} = [\max(\text{load}_t^{\uparrow}(s), \text{load}_t^{\downarrow}(s)) > U_s], \tag{4.1a}$$

$$R_{s,t} = (1 - c_{s,t}) \frac{g(\text{load}_t^{-}(s))}{traffic_s} - c_{s,t}, \tag{4.1b}$$

replacing $\text{load}_t^{-}(s)$ in eq. (4.1) with whichever load direction is prioritised according to the traffic characteristics of the deployment environment. $c_{s,t}$ represents an 'overloaded' condition at destination s, equalling 1 if either load for s is greater than its capacity. We choose $\text{load}_t^{\uparrow}(\cdot)$ for defending UDP-based models and $\text{load}_t^{\downarrow}(\cdot)$ for HTTP traffic, though $\text{load}_t^{\updownarrow}(\cdot)$ would likely be the most suitable for general deployment or heterogeneous traffic patterns. These choices reflect where the bulk of transmitted bytes in each traffic model is observed—and the lack of this knowledge in the general case.

While the use and definition of $g(\cdot)$ appears nebulous, there are many possible ways to infer this quantity in practice. End-host servers may use ca-

nary flows or other active measurements, or employ existing QoE metrics in the case of VoIP services such as lost packets, reorderings, and jitter. ASes and transit networks may make use of reports received from downstream networks, e.g., over the DOTS protocol (Dobbins *et al.*, 2021). Even if such heuristics or perfect knowledge aren't available in deployment, a sufficiently well-trained agent needs only to greedily follow the policy it has learnt from training, allowing pre-training by a simulated environment (with perfect knowledge) to transfer to reality.

If a network is believed to be vulnerable to indirect attacks, such as link-flooding attacks, we may use the following reward:

$$R_{s,t}^{Cross}(\beta) = \beta R_{s,t} + (1-\beta)\min\{R_{s',t} \mid s' \neq s\} \tag{4.2}$$

where the collaboration parameter $\beta \in [0,1]$ models the expected degree of interference between flows, and s, s' are protected destination nodes in the network. The key insight underpinning LFAs is that flows can affect a target *without communicating with that target.* β then acts as a tunable parameter which can incentivise agents to remove flows which harm overall system health, by including the performance of the worst-performing destination. However, such attacks (and the effectiveness of $R_{s,t}^{Cross}$) are not examined by this work.

4.4.5 Action space

When monitoring a source-destination pair, an agent uses its state vector to decide which proportion of that flow's *inbound* traffic should be dropped. This is implemented by installing an action via OpenFlow, instructing its host switch to drop each relevant packet with probability p. Although it invites risks which I describe shortly, agents choose to drop packets rather than impose traffic limits as it offers a discrete action space without prior knowledge of traffic characteristics or measurement. Furthermore, we need not consider burstiness, fairness, or tuning of queue parameters (such as per-flow bucket sizes) which could limit scalability. I present two models on how to choose p: *Instant* agents which directly choose p over a uniform domain, and *Guarded* agents which follow a reduced action set controlling an FSM.

Instant control Each agent directly chooses $p \in \{0.0, 0.1, \ldots, 0.9\}$, giving a discrete, static action set which cannot completely filter traffic. These choices ensure that the rate reduction imposed on traffic from a given source IP may never be permanent or irreversible. Since this model needs no forward planning, I found it best to set the discount factor $\gamma = 0$ (making agents purely myopic).

Guarded control The measurements of Kang *et al.* (2016b) suggest that bot attack flows cannot scale up to match an increase in available bandwidth. I apply their observations within the RL paradigm by constraining how an agent treats each flow using a simple FSM: we restrict $p \in \{0.00, 0.05, 0.25, 0.50, 1.0\}$. The action set is then simply to *maintain*, *increase*, or *decrease* p for a flow in single steps. We choose these potential values for p to add complete filtering to a steady progression of rate-limiters (25 % increments for congestion-unaware UDP traffic). The outlier, $p = 0.05$, corresponds to roughly a 50 % rate reduction for congestion-aware TCP flows in our test topology. This uneven spread of choices for p allows light and heavy rate reduction to be applied to both congestion-aware and congestion-unaware traffic as required.

To enable temporary bandwidth expansion in all deployments, every flow is initially placed under significant (but still somewhat usable) packet drop ($p = 0.05$); this is chosen above the equivalent for UDP due to TCP's higher prevalence. Most importantly, an agent must now choose to punish a flow multiple times in succession to cause rapid degradation, reducing variance while allowing an agent to see how a host reacts to structured changes in the environment.

As each agent now requires the capability to plan ahead, we require a discount factor $\gamma \neq 0$, allowing the value of future states to influence state-action value updates. Setting $\gamma = 0.8$ was found to be the most effective choice for this hyperparameter during exploratory testing.

Risks This mode of action means that each agent is in control of push-back (Mahajan *et al.*, 2002), and so carries a risk of introducing collateral damage into the network. Recall from the critical analysis of *Marl* in section 3.1.3 that benign congestion-aware traffic which responds to packet loss as a congestion signal will *explicitly slow down further*. This is of particular importance due to the prevalence of TCP and other congestion-aware protocols within the Internet. Analysis of the CAIDA datasets for 2018 (CAIDA, 2018) shows that congestion-aware traffic makes up at least 73–82 % of packets, corresponding to 77–84 % of data volume (appendix A). The QUIC transport protocol, which has become commonplace, is congestion-aware and makes up around 2.6–9.1 % of traffic observed on backbone links, depending on location and typical workload (Rüth *et al.*, 2018). Making the wrong action choices here will have a greater impact on most legitimate traffic.

Choosing the wrong granularity to apply actions can be similarly disastrous. The other key weakness of Marl is that actions are applied on a per-switch basis. This further justifies our focus on per-flow decisions—real-world deployments see many flows pass over any egress point, making such global actions more likely to inflict collateral damage. We can show analytically that the intuition that per-flow decisions provide better service to carried traffic holds in this case. Given the probability that a host is legitimate,

$P_G \in [0, 1]$, it follows that a host will be malicious with probability $P_B = 1 - P_G$. Defining *imperfect service* to mean any case where all n hosts whose traffic is carried by a single switch do not share the same classification (i.e., a mixture of benign and malicious hosts), then the probability that a switch is delivering imperfect service is $P_{M,n} = 1 - (P_G^n + P_B^n)$.

Theorem 1. *As the host-to-learner ratio n increases, it is more likely that a throttling switch will exhibit imperfect service: $\forall n \in \mathbb{Z}^+, P_{M,n} \leq P_{M,n+1}$.*

Proof. Base case: $P_{M,1} = 0, P_{M,2} = 1 - P_G^2 - P_B^2 > 0$.

Inductive step: Assume that the theorem holds for n. Observe that $P_G^n \geq P_G^{n+1}$ (resp. P_B). It then follows that:

$$P_G^n + P_B^n \geq P_G^{n+1} + P_B^{n+1}$$
$$1 - (P_G^n + P_B^n) \leq 1 - (P_G^{n+1} + P_B^{n+1})$$
$$P_{M,n} \leq P_{M,n+1} \square$$

Corollary 1.1. *Restricting $P_G \in (0, 1)$ so that both P_G and P_B are non-zero ensures strict inequality: $P_{M,n} < P_{M,n+1}$.*

When considering that many hosts have an especially adverse reaction to our main means of control, flow-level granularity becomes an obvious choice.

4.4.6 Systems considerations

Acting on an unbounded set of flows in each timestep introduces potential issues: the inability to respond to unexpected changes in flow state, delayed service of new flows, and the risk that flow states become outdated. At their worst, these risks present additional attack surface to an adversary. To tackle these problems, we make use of *Timed Random Sequential* (TRS) updates.

The scheduler begins with a shuffled work list of active flows. When requested, the scheduler estimates the cost of action inference using past timing information, and proceeds down the list to send a set of flow 5-tuples to the core Sarsa logic which can be handled in a set time limit. The scheduler continues until the list is empty, at which point it is repopulated and reshuffled with active flows.

Following the observations of L. Chen *et al.* (2018) concerning short flows, we maintain a deadline of 1 ms—in tests, an agent is typically able to process around 3 flows in this time. Ideally, deadlines should be tuned based on the frequency at which statistics arrive. Naturally, an agent must carry work forward (and coalesce state updates) when *host density* is $n > 3$ (section 4.8); this behaviour is not explicitly a property of network size, but relates to the cost of inference and learning. The amount of processed flows per deadline

depends on the agent design (required FPU operations, policy size), but also on the amount of flow telemetry data needing processed—the current implementation is written in Python, restricting this handling to a single thread. An implementation in a systems language such as Rust or C++ would allow faster concurrent processing.

There is a risk that so much work can be queued up that an agent is never able to act on some attack flows. A solution is to impose an upper bound on the amount of action inference and policy updates that can be performed before the work list is regenerated. This removes the guarantee that all flows will be visited often, but if updates occur regularly then this sampling may be sufficient to achieve good performance.

4.5 System architecture

To demonstrate how we would measure state throughout the network, coalesce it, and act upon flows in an effective way, I present the design of a system which supports the effective real-world deployment of the above RL agents. Figure 4.2 displays this, separating system elements which are local to each agent from those which reside elsewhere in the network. Each agent here operates as a VNF adjacent to a software-defined switch, to which it acts as an extra controller for installing install actions, forwarding rules, and logic into a switch at runtime. Agent VNFs communicate with these co-hosted switches to install probabilistic packet-drop rules. I describe the main purpose and operation of each module within an agent's VNF, and discuss techniques to make deployment more efficient using existing technologies.

4.5.1 Core and RL executor

The core module is the main loop in an agent's architecture. At each timestep, the core receives information about which flows have arrived and should be acted upon from the *TRS Scheduler*. The core then retrieves the current and previous state vector associated with each flow from the *Flowstate Database*, passing them into the RL algorithm alongside the last action chosen for that flow (if available).

The RL algorithm then infers and returns an action. Each action is passed to the database, which computes and returns a packet drop rate according to the agent model (*Instant/Guarded*) while updating flow state. This is then converted into an OpenFlow message carrying packet drop rules; these are batched to the agent's switch using the same groupings produced by the scheduler. Finally, timing information is passed back into the scheduler to refine its estimates about how much work should be scheduled in the next timestep.

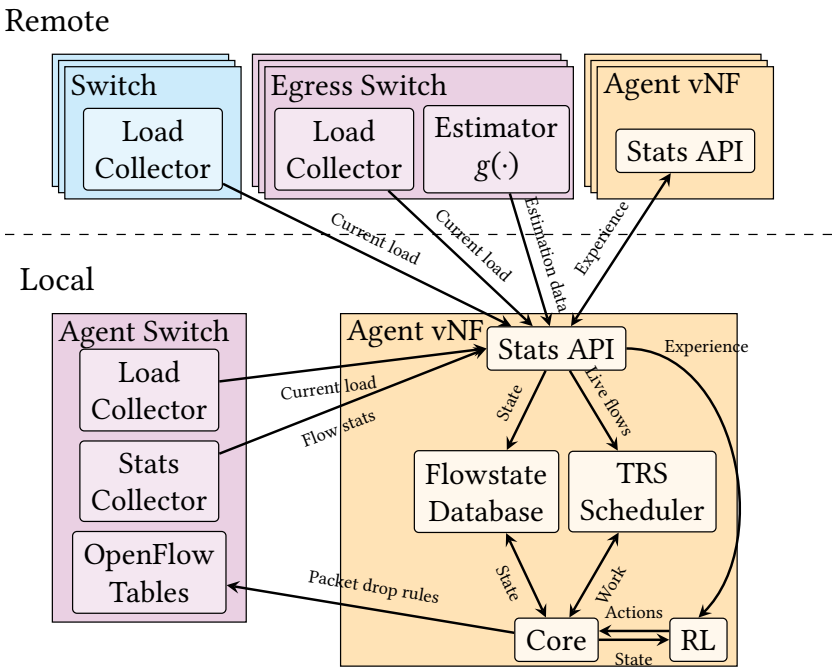


Figure 4.2: VNF and OpenFlow-based system architecture for an RL-driven DDoS mitigation system. Each edge node of the network runs a single agent, using an OpenFlow switch to probabilistically filter traffic, with a co-located host machine used to perform RL inference and learning. Estimators using, e.g., DOTS information to estimate legitimate traffic communicate with these edge switches, which also share RL trace data with one another to learn collaboratively. Most switches in the network need only to report their load statistics to a shared controller. As this is a common capability, this architecture can serve a network of mostly legacy switches.

State space sizes guarantee that an *Instant* policy remains under 520 KiB, though the sparse representation admitted by tile-coding typically leads to far smaller policies of around 17.8 KiB from my experiments. *Guarded* policies are 30 % of this size. As described earlier, action updates require a constant number of floating point operations, and the vast majority of these operations can be vectorised. Action computation for *Guarded* agents is cheaper still, on account of having fewer actions.

4.5.2 Stats API and collectors

Agents require information from the network and one another to be effective. These agents can act either independently, having no agent-to-agent communication, or cooperatively. In the latter case agents transfer, when possible, *experience* to one another—lists of state-action-state transitions with associated rewards. It's noted that a transition may be high-value or surprising to one agent, while well-known to another, causing each to produce different policy updates from the same unit of experience. For this reason I do not transfer policy deltas between agents, causing each to learn its own policy. Determining which scheme achieves better performance is, however, left to future work.

Load collectors and estimators periodically push observations to each active agent VNF. In the current implementation, load statistics are gathered via VNFs active at each network switch, though we expect that OpenFlow stats requests, NetFlow or SNMP data may be used to derive these cheaply. Transferring state to agents and experience sharing can both be made more efficient through effective use of broadcast addressing in a target network. Depending on the capabilities of switches in the network, the estimator can either send benign traffic estimates or parameters for use in a reward function.

Gathering and transmission of load/flow statistics would be difficult to perform quite as often as an emulated environment allows. However, the measurements acquired in such a scenario are likely to be less noisy (by being collected over longer periods of time), which could aid effective training.

4.5.3 Flowstate database

For each flow 5-tuple, we hold two state vectors containing the features described in section 4.4.3—the current state, and the state which induced the last action. To ensure that flow control actions are made with recent information, we combine state vectors for unvisited flows in the current work set. State vector combination is done by coalescing and combining state vectors as described in section 4.4.6. For flows outside of the current work list, we simply replace or insert the new state vector.

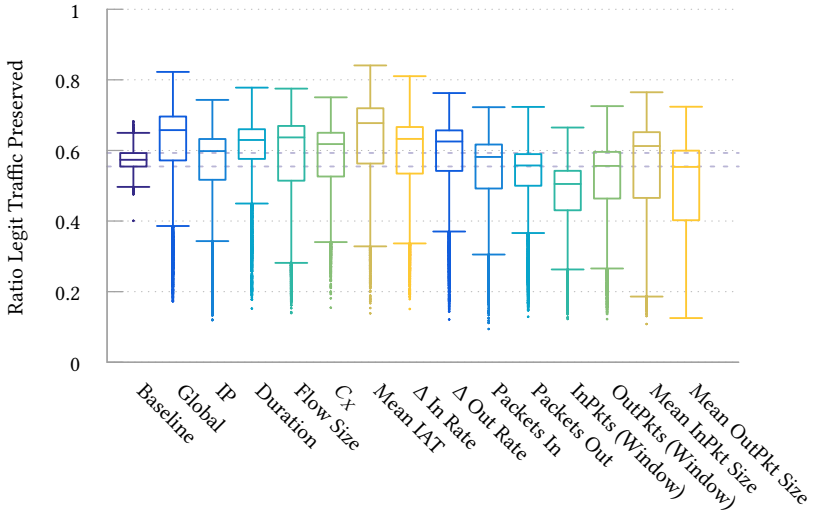


Figure 4.3: Learnt performance of *Instant* agents when benign traffic is UDP-like, using only a single feature as a basis for decisions. Mean IAT, inbound packet sizes, and global state offer the best predictive performance, while most features offer marginal advantage over the unprotected baseline.

4.5.4 Agent switches

Agent switches operate a modified version of OVS, implementing an action which requests that each matched packet be dropped with a certain probability. Ideally, this host-based solution would be replaced with a bespoke ASIC running OpenFlow with this extension. To get around the lack of floating-point support in many environments, such as PDP environments or OVS's kernel datapath, I represent this probability using a 32-bit unsigned integer (scaling 1.0 to $2^{32} - 1$). On commodity hardware without explicit packet drop support, I believe that a similar effect can be achieved using OpenFlow meters (at the expense of these being stateful measures).

OpenFlow groups are used to simplify control messages: premade tables with permitted levels of packet drop. This saves some overhead compared to using experimenter/extension headers. Flows are automatically given a group with the default level of packet drop (according to the chosen agent design), meaning that switches don't need to refer to a controller or the agent VNF.

4.6 Rethinking the state space

The main element required by a per-source model is a feature set with high predictive power, so that behavioural differences are apparent to an agent. Elaborating on the statistics discussed in section 4.2 which others

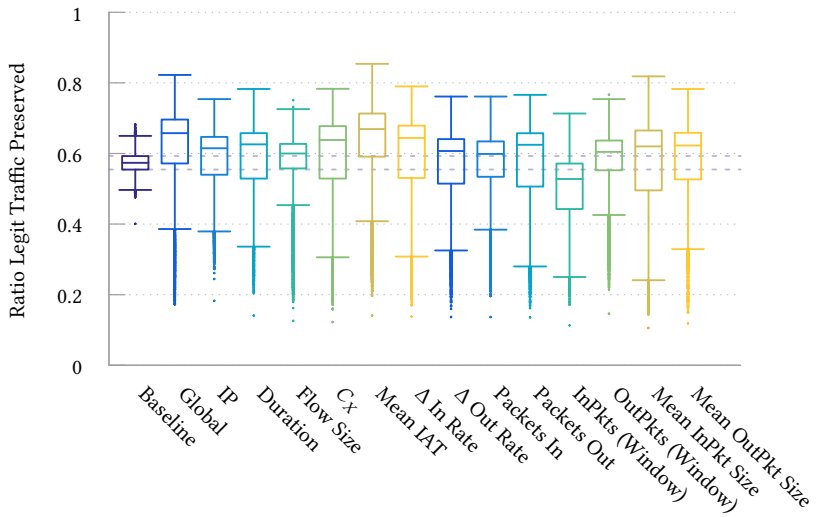


Figure 4.4: Learnt performance of *Instant* agents when benign traffic is UDP-like, combining each feature with the last action taken as a basis for decisions. Compared to fig. 4.3, this combination causes a marked improvement in the packet count and per-window statistics, and leads to a tighter performance bound for Flow Size.

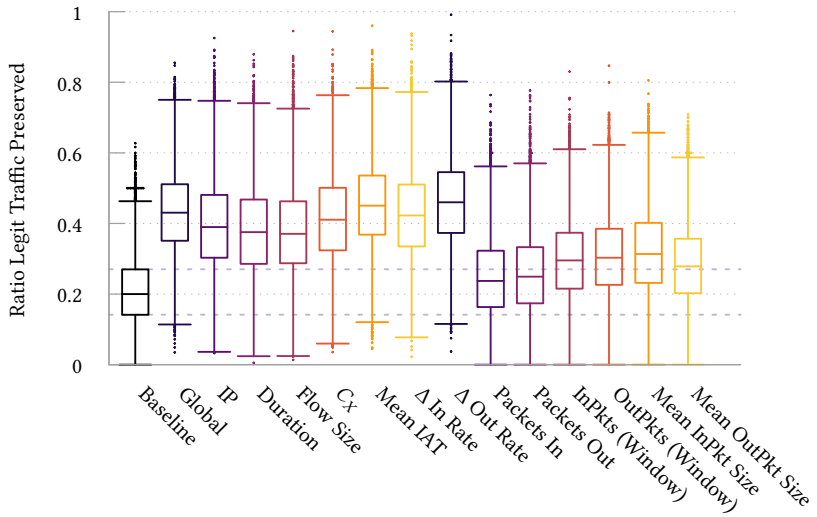


Figure 4.5: Learnt performance of *Instant* agents when benign traffic is TCP-like, using only a single feature as a basis for decisions. All of the chosen features can offer a marked improvement over no protection at all. Global state and Mean IAT still offer the greatest improvement above baseline, but packet-level statistics are considerably less effective for this class of traffic.

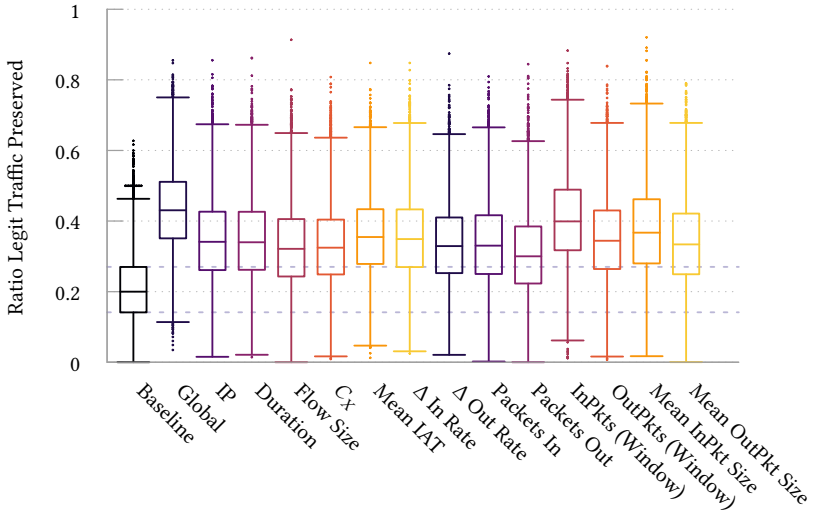


Figure 4.6: Learnt performance of *Instant* agents when benign traffic is TCP-like, combining each feature with the last action taken as a basis for decisions. This combination causes a significant improvement in the effectiveness of packet-level and per-window statistics.

have shown to be effective, I believe the following features to be useful (and humanly justifiable), and investigate their use alongside different traffic types.

Global state This is the vector of load measurements along a flow's path introduced in section 4.4.3. These values indicate the overall health of the network, and crucially are all measurements which an agent has some degree of direct control over—though the inputs of all agents on all flows are aggregated for later load measurements.

Source IP address While trivial to spoof (and thus of limited use for many classes of attack), reflectors are themselves legitimate services being abused by spoofing attackers. As a result, they communicate with attack victims using their own IP address. In real-world scenarios the addresses of reflector nodes might exhibit similarity due to network uncleanliness (Collins *et al.*, 2007), e.g., unhardened services exposed by a single organisation.

Last action taken This encodes an agent's current belief in the maliciousness of a flow. This feature also potentially allows forgiveness, serving as a reference point for determining whether a source mistakenly marked as malicious exhibits different falloff behaviour after punishment. It's important to note that this feature only makes sense once combined with another flow feature, and never appears individually tile-coded.

Flow duration and size Features which describe the length of time a connection has been active, and the amount of data transferred within that time. An extraordinarily long flow, having sent a lot of data, could be more likely to be an amplifier: though most (62 %) waves of amplifier traffic last shorter than 15 min (Krämer *et al.*, 2015), this is considerably longer than the typical length of an HTTP request/response.

Correspondence ratio The ratio between upstream and downstream traffic associated with a source IP address. I define this here as:

$$C_X = \min(\text{load}_t^{\uparrow}(\cdot), \text{load}_t^{\downarrow}(\cdot)) / \max(\text{load}_t^{\uparrow}(\cdot), \text{load}_t^{\downarrow}(\cdot)),$$

where a value close to 0 indicates strong asymmetry.

***Δ* Send/receive rate** The change in traffic rates caused by the last action. Behavioural changes induced by bandwidth expansion or reduction are expected to be most visible here.

Mean inter-arrival time A measure of how often packets arrive at an agent's parent switch; low IATs indicate a high number of packets per second, and can be a possible marker of malicious behaviour. I only make use of the mean IAT of *inbound* traffic.

(Per-window) packet count The amount of packets sent to/from a source over a flow's lifetime (or the current window of measurement), similar in use to flow size and mean IAT.

Mean packet size per window Legitimate flows, both congestion-aware and -unaware, often transmit packets with a distribution of sizes. Attack traffic is not likely to be so diverse: we might expect solely max-size packets in the case of amplification attacks, or minimum-size packets in other flooding attacks.

The exclusion of features such as source/destination ports or protocol numbers is a deliberate choice. If QUIC or a similar protocol were to become ubiquitous, then these fields would have little to no correlation with the class of traffic a flow might contain. My aim was to design around this constraint as a form of future-proofing.

All of the above features, save for global state, are 1-dimensional. For simplicity, here 'UDP' refers to congestion-unaware traffic, while 'TCP' refers to congestion-aware flows. Figure 4.3 shows the effectiveness of each feature for UDP (resp. fig. 4.5 for TCP), on a single-destination topology (section 4.8.1) with $n = 2$ hosts per egress point averaged over 10 runs. Figure 4.6

Table 4.1: Tile coding windows for traffic features.

New Feature (unit)	Range
Load (Mbit/s)	$[0, U_s]$
IP	$[0, 2^{32} - 1]$
Last Action (%)	$[0, 1]$
Duration (ms)	$[0, 2000]$
Size (MiB)	$[0, 10]$
Correspondence Ratio	$[0, 1]$
Mean IAT (ms)	$[0, 10\,000]$
ΔIn/Out Rate (Mbit/s)	$[-50, 50]$
Packets In/Out	$[0, 7000]$
Packets In/Out Window	$[0, 2000]$
Mean In/Out Packet Size (B)	$[0, 1560]$

demonstrates how feature accuracy varies when tiled alongside *last action*, with similar trends observed when applied to UDP traffic (fig. 4.4). The plots show that different protocols and traffic classes are best defended by different features—as such, every feature presented has value in a complete model. All features converge to their highest-observed performance within around 4000 timesteps. In general, some of the most effective features are the global state, mean IAT, mean inbound packet size and Δ rates.

4.7 Traffic modelling

Here, I describe network models built around live testing of reactive TCP and UDP traffic in an SDN-enabled environment, which is adaptable to arbitrary topologies, with an explicit focus on preserving their real-time dynamics in a way that trace-based evaluation cannot. First and foremost, the goal is to replicate representative load and packet inter-arrival characteristics, and to capture how these characteristics evolve in response to actions. I introduce these models because we are interested in capturing interactive, correlated back-and-forth exchanges associated with live HTTP traffic; mainly because of the particular interactions between the application-level dynamics, congestion awareness at the transport level and the nature of control signal used.

4.7.1 Network design

We make use of a fully software-defined network, built using OpenFlow-aware switches in *mininet* (Mininet Project, 2022) alongside a controller based on *Ryu* (Ryu SDN Framework Community, 2017). All internal routers are primed with knowledge of the shortest path to each internal host, while new inbound flows register the 'way back' for each hop used, to ensure consistent bidirectional traffic conditions for each flow. If several ports offer

different (equal-length) paths to a destination, a consistent random port is chosen from the flow-hash by an OpenFlow *Group action* (in *select* mode). If such information is lost, perhaps expiring due to inactivity, it suffices to forward an outbound packet on a random outbound port, as we assume that any external IP address is reachable through any of the test network's egress ports (i.e., that it is not connected to any stub ASes). The controller is also responsible for computing how switches respond to ARP requests: this need arises due to the reliance upon Linux's networking stack for live applications, and wouldn't need to be considered for trace-based evaluation.

4.7.2 TCP (HTTP) traffic model

To model legitimate TCP traffic, server nodes run an nginx v1.10.3 HTTP daemon, serving statically generated web pages alongside various large files and binaries. Benign hosts run a simple libcurl-based application written in Rust, repeatedly requesting resources from the server. Hosts and clients both use TCP Cubic (Rhee *et al.*, 2018). Each host's download rate is limited to match the maximum bandwidth assigned to it, and requests several random files known to exist within a website, followed by any dependent resources for each (stylesheets, images, etc.) as a browser might. On completion, a host changes its IP address to generate separate statistics per-flow, while minimising downtime. This presents a balanced distribution of flow duration and size, with large files included to model elephant flows.

4.7.3 UDP (Opus/VoIP) traffic model

VoIP traffic exhibits very different characteristics to the above model; packet arrivals are highly periodic due to real-time requirements, flows have a constant bitrate, and do not react substantially to lost packets. Interestingly, DDoS attack traffic is known to share many of these characteristics, offering an interesting detection problem. I present a VoIP traffic model based on Discord (Discord, 2022), a freely-available messaging and VoIP platform geared toward gaming communities. Discord is a good model for this prototype due to its publicly documented API, many open source bot frameworks, large user base, and due to the lack of models for Opus-encoded traffic. Further details on trace measurement and generation are provided through appendix B.

Hosts send *Real-time Transport Protocol* (RTP) traffic with Salsa20 encrypted payloads—20 ms audio frames at 96 kbit/s. We generate similar traffic at hosts by replaying anonymised traces gathered in general use and tabletop role-playing servers; each trace contains only the size of each audio payload, entries denoting missed packets, and the duration of silent periods. We trim these silent periods to a maximum 5 s due to the lengthy talk/silence bursts introduced by users in RPG servers, and estimate the size of missed

packets by taking an exponentially-weighted moving average over known sizes. Hosts punctuate audio frames with a 4-byte keepalive every 5 s. All traffic passes over a central server which groups hosts into rooms, and is forwarded to other participants; we do not replicate pre-call Websocket traffic which would be used for authentication. There is no peer-to-peer traffic—the server acts as a *Traversal Using Relays around NAT* (TURN) relay for all hosts. Each flow occupies an expected 52.4 kbit/s upstream bandwidth. To match the target upload rate assigned to a host, each runs enough individual sessions to meet the target data rate.

4.7.4 Attack traffic model

Malicious traffic is generated by use of the *hping3* program, generating UDP-flood traffic targeting random ports. Each instance of *hping3* was configured to generate Ethernet MTU-sized packets (1500 B) with a random source and destination port towards a target server, and configure the output rate r (in Mbit/s) by setting the inter-arrival time $t_{attack} = \frac{1500 \cdot 8}{r \cdot 10^6}$. This fulfils certain characteristics of many types of amplification DDoS traffic: it is congestion-unaware (Rossow, 2014), and packets are larger than the minimum frame size and identically-sized. This latter behaviour is seen in the wild: NTP amplification traffic is fragmented at the application layer into 482 B chunks (Cisco, 2014). This model differs from NTP amplification in frame size so that inter-arrival times are larger, to keep emulation of the network feasible at high traffic rates.

4.8 Evaluation

This work is most naturally compared against Marl, introduced by Malialis and Kudenko (2015), the state-of-the-art in RL-based DDoS prevention. We are most interested in seeing how their approach contrasts with the new agent designs across different topologies and workloads. Different network environments will also impose different levels of host density, where popular web servers may have orders of magnitude more clients than egress points from their network—I aim to show how these characteristics affect performance and learning rate. Marl is known to outperform the AIMD (Yau *et al.*, 2005) strategy, yet the state of the art has long since moved on. To paint a more current picture, I compare this work against an effective modern approach, *SPIFFY* (Kang *et al.*, 2016b). SPIFFY tests a proportion of flows by routing them through an alternate path with higher bandwidth, observing how their speed changes some time later. This comparison lets us position our new agent designs against the state of the art, observing that SPIFFY has a similar mode of interaction to RL-based systems (taking action, observing an effect, and acting once again) and does not rely on protocol characteristics or signatures. In reimplementing SPIFFY, I make the

simplifying assumption that a suitable unused path exists (with identical bandwidth to the server's link). 10 % of active flows were tested at a time (according to the authors' observation that there is a factor of 10 × difference between the ideal and achieved bandwidth expansion), excluding flows below 50 kbit/s and requiring a 3 × expansion from legitimate flows, making a judgement after 5 s.

To test this, I made use of both traffic models introduced in section 4.7 (Opus and TCP), both topologies discussed below (1-dest vs. Fat-Tree), and vary the amount of hosts typically communicating over each agent's ingress/egress node. Additionally, these new models were evaluated in multi-agent mode (*separate*, no model sharing), and in single-agent mode (*single*, zero-cost perfect information sharing). In each case, the algorithm's performance was averaged over 10 episodes of length 10 000 timesteps (setting each agent's $\boldsymbol{\theta} = \mathbf{0}$ between episodes). Host allocations at the beginning of each episode were generated pseudorandomly to ensure fairness between episodes—a host is malicious with probability P (*malicious*), and is benign otherwise. Benign hosts generate traffic according to either sections 4.7.2 and 4.7.3 depending on the experiment, while malicious hosts generate traffic as described in section 4.7.4 (both at experiment-dependent rates).

All experiments were executed on Ubuntu 18.04.2 LTS (GNU/Linux 4.4.3-040403-generic x86_64), using an Intel Core i7-6700K (4 × 4.2 GHz) which had 32 GiB of RAM.

4.8.1 Single destination

The network is tree-structured, where one server s connects through a dedicated switch to k team leader switches, each connected to ℓ intermediate switches, which in turn each connect to m egress switches. We then have $N_{hosts} = k\ell mn$. Figure 4.7 demonstrates this. The network topology was configured using $k = 2$ teams, $\ell = 3$ intermediate nodes per team, $m = 2$ agents per intermediate node, and $n \in \{2, 4, 8, 16\}$ hosts per learner. This is a slight simplification of Malialis and Kudenko (2015)'s 'online' experiment, choosing fewer teams but remaining as a single server with a fan-out network.

4.8.2 Multiple destinations

The previous topology allows for direct comparison against the state-of-the-art, and indeed is illustrative of one way in which attack traffic might aggregate in the network. It is hard, however, to argue its relevance to specific classes of victim or to reason about the interactions it might have with dependent applications. In contrast, the fat-tree topology (Al-Fares *et al.*, 2008) sees regular use in real-world data centres and scales well horizontally. We

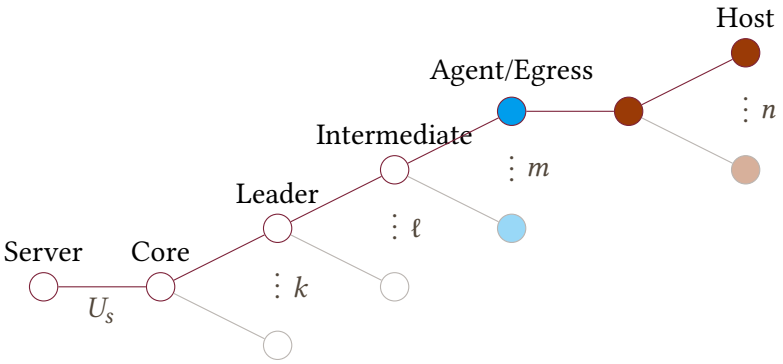


Figure 4.7: Network topology diagram, showing how the server and its core switch's k teams are structured, with ℓ intermediate routers per team, connected to m agents which each moderate n hosts beyond a single external switch. Red nodes are external, and each blue node hosts an agent.

use a $k = 4$ fat-tree, with one pod hosting two servers s_0 and s_1. n external hosts connect through each core switch (where agents are hosted), and communicate with s_0, s_1 uniformly randomly. Both servers host identical services. We set $n \in \{6, 12, 24, 48\}$ hosts per learner (keeping N_{hosts} identical to each tier of the single-host topology), and restrict $U_{s_0} = U_{s_1} = U_s/2$.

4.8.3 Parameters

The algorithm parameters were set at $\alpha = 0.05$, linearly annealing $\epsilon = 0.2 \to 0$ by $t = 3000$ in the case of Marl (8000 actions per agent in the *Instant/Guarded* models).

Benign hosts each occupied 0–1 Mbit/s, and hosts were redrawn at each episode's start with P(*malicious*) = 0.4. Malicious hosts each sent 2.5–6 Mbit/s when attacking UDP traffic, though this was increased to 4–7 Mbit/s when using TCP-like traffic (to meaningfully impact benign flows). Given n and P(*malicious*), we see an expected malicious bandwidth 1.27–1.87 and 2.03–2.18 × U_s respectively. For these choices of n in both topologies, we observe $N_{hosts} \in \{24, 48, 96, 192\}$, and an expected number of malicious hosts $\mathbb{E}[N_{attackers}] \in \{9.6, 19.2, 38.4, 76.8\}$. For the largest choice of n, we see an expected total attack traffic $\mathbb{E}[V_{attack}] = 334.05$ Mbit/s and 422.4 Mbit/s for Opus and HTTP traffic respectively.

U_s was fixed at $N_{hosts} + 2$ Mbit/s (to account for burstiness), and each link had a delay of 10 ms. All links had unbounded capacity, save for each server-switch. These parameters match those of the original study to enable direct comparison, and many are (to the best of our knowledge) arbitrary, but I justify the range of n as capturing increasing scales of host activity.

Table 4.2: Average reward for combinations of model, host density and traffic class with a single destination.

Traffic	n	SPIFFY	Marl	Instant		Guarded	
				Separate	Single	Separate	Single
OPUS	2	0.043	0.628	**0.629**	0.448	0.430	0.629
	4	0.069	0.538	**0.653**	0.449	0.308	0.571
	8	0.065	0.468	**0.533**	0.516	0.398	0.507
	16	0.053	0.460	0.438	0.452	0.347	**0.504**
TCP	2	**0.799**	0.305	0.061	0.068	0.241	0.196
	4	**0.953**	0.359	0.191	0.097	0.278	0.504
	8	**0.995**	0.362	0.376	0.201	0.357	0.605
	16	**0.999**	0.320	0.316	0.302	0.478	0.708

Table 4.3: Average reward for combinations of model, host density and traffic class with multiple destinations.

Traffic	n	SPIFFY	Marl	Instant		Guarded	
				Separate	Single	Separate	Single
OPUS	6	0.092	**0.382**	0.300	0.170	0.307	0.189
	12	0.096	0.217	0.322	0.275	**0.333**	0.235
	24	0.125	0.404	0.358	0.296	0.382	**0.461**
	48	0.110	0.430	0.418	**0.438**	0.427	0.428
TCP	6	**0.692**	−0.222	0.123	−0.018	0.121	0.116
	12	**0.896**	0.008	0.132	0.008	0.163	0.266
	24	**0.974**	0.063	0.130	0.024	0.337	0.390
	48	**0.995**	0.156	0.219	0.111	0.431	0.499

4.9 Results

We now examine the performance of the two new models (*Instant*, *Guarded*) as compared against existing RL work (*Marl*) and *SPIFFY* under different traffic behaviour and topologies, varying the host-to-learner ratio n and environment. Tables 4.2 and 4.3 present the average rewards for all combinations of these factors—providing a rough idea of expected performance, with the highest-performing model in bold and the best RL-based model underlined. Average rewards take into account any portions of time that an agent allows illegal system states. Several plots augment this, illustrating peak performance or the amount of time which an agent requires to learn.

4.9.1 Congestion-unaware traffic

In a single-destination network, we observe that Marl's performance degrades as n increases. Typically, our *Instant* agent design achieves the best performance in multi-agent mode, having lower collateral damage than the

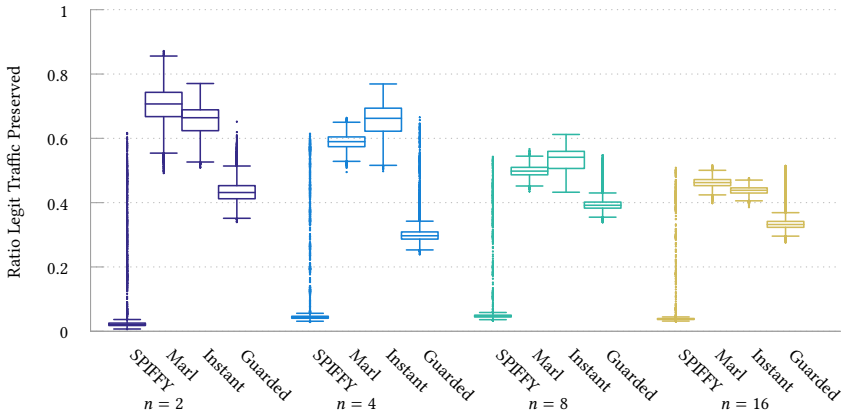


Figure 4.8: Online performance for Opus benign traffic in a single-destination network, multi-agent mode. *Instant* outperforms Marl for $n \in \{4, 8\}$ (with higher variance), but performs similarly to Marl at $n \in \{2, 16\}$. *Guarded* underperforms compared to the other agent designs in this problem variant.

current state-of-the-art, but sharply degrades at low n when agents share experience. This trend reverses for the *Guarded* model, which improves as n increases and in single-agent mode—when $n \geq 4$, the single-agent variant offers consistent improvement. Figure 4.8 shows the preserved traffic in multi-agent mode. When defending multiple destinations, we see a sharp decrease in the effectiveness of all agent designs. The new *Instant* and *Guarded* agent designs become more effective as n increases, while Marl's effectiveness is roughly constant (aside from the outlier at $n = 12$). Interestingly, SPIFFY is unable to effectively protect CBR traffic.

4.9.2 Congestion-aware traffic

Table 4.2 shows that Marl offers a low (though fairly consistent) level of protection for TCP traffic, which the *Instant* agent offers no substantial improvement over. However, *Guarded* agents offer a remarkable improvement for this class of traffic, particularly when experience can be shared—offering a 2.21 × improvement over the state-of-the art during training, which is made clearer in fig. 4.9. Figure 4.10 shows that this model can protect a peak 80 % of TCP traffic (2.5 × improvement) after just 100 s, but also that all of the new models require considerably longer than Marl to learn their best-achieving policy. We observe that the same trends present themselves in the multi-destination topology: *Guarded* remains the best fit for TCP, in both training modes. Crucially, the rigid tree of learners and teams which define Marl, along with its lack of action granularity, seem to be a poor fit in this environment. In both cases, SPIFFY greatly outperforms the RL-based methods.

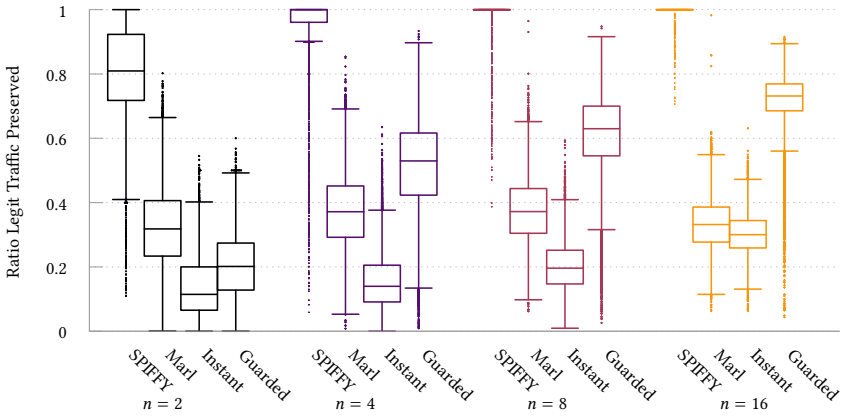


Figure 4.9: Online performance for HTTP benign traffic in a single-destination network, single-agent mode. *Instant* and *Guarded* exhibit similar efficacy at $n = 2$, protecting less traffic than Marl. Only *Guarded*'s performance rapidly increases with n, achieving a considerably better median and lower variance than the other models. The longer tails of outliers typically indicate the longer training time the new models require—we observe that *Guarded* typically has considerably lower variance once it has converged on a stable policy.

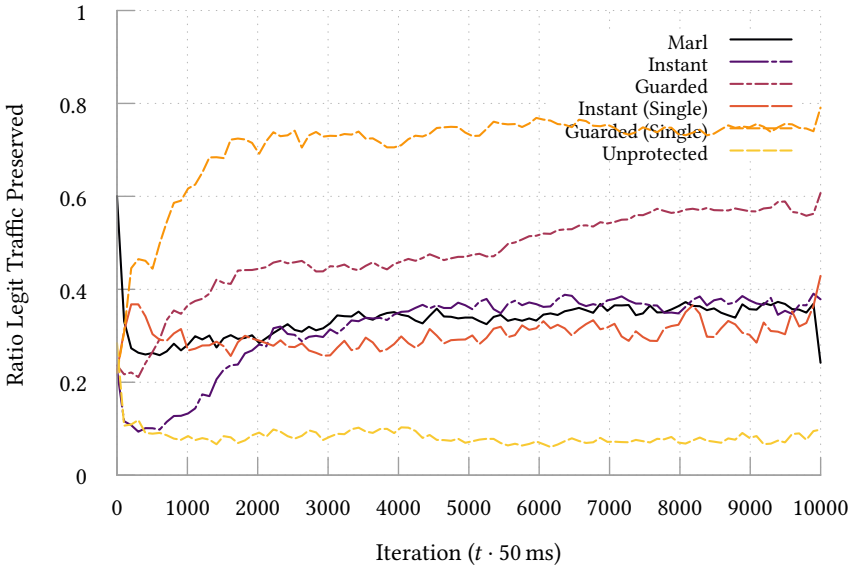


Figure 4.10: Online performance of standard and single-agent models in a single-destination network with $n = 16$ hosts per egress point, HTTP traffic. At this level of host density, *Guarded* reaches higher peak performance sooner and is considerably more consistent throughout the episode. *Guarded* benefits greatly from information sharing, converging to protect around 75 % of TCP traffic within 100 s. The *Instant* model converges to Marl's level of performance.

Table 4.4: Average reward versus attack volume.

Factor	$\mathbb{E}[V_{attack}]$ (Mbit/s)	Reward
1.5	633.6	0.671
2.0	844.8	0.625
2.5	1056.0	0.620
3.0	1267.2	0.619
3.5	1478.4	0.600

4.9.3 Increased attack volume

To assess the effect of larger volumes of attack traffic, we increase an attacker's output by various factors, supposing $n = 16$ with HTTP traffic (*Guarded*, Single); table 4.4 records the expected rate of attack and average performance. The initial increase in traffic volume causes the steepest reduction in performance (due to the increased cost of incorrect action), though performance levels out as attack traffic increases.

4.9.4 Computational cost

Measurements from each of these experiments indicated that the cost of computing any action is takes a median 515.94 µs (following the Float results for *Collector* in table 5.2), and is typically 80–100 µs per flow during a decision narrowing. This is reassuring when measured alongside the insights from other work. L. Chen *et al.* (2018) observe that, ideally, actions must be computed and taken within 1 ms to have a meaningful effect on short flows. That our starting point falls significantly below this threshold allows us to safely consider more costly actions or larger state spaces, which would typically increase the computational cost. This cost is constant and independent of network size. As discussed in section 4.4.6, we are able to judge 2–3 flows before this deadline, dependent on ϵ: this is worsened in practice by serialisation & communication delays, learning logic, and single-threaded processing in the Python language.

4.10 Discussion

Model performance Of the results presented, *Guarded*'s unpredictable (often worse) starting performance is unexpected, given its far smaller action space. It's natural to expect that this would make the model easier to learn, but the additional state required appears to make the task *harder*, beyond even the value of choosing a non-zero discount factor—which added forward-planning to explicitly mitigate this effect. Accordingly, we see that this design performs best (and exhibits considerably lower variance) when agents learn from as much knowledge as possible: high n and single-agent training.

To filter incoming traffic from a source, it must decide to degrade inbound traffic multiple times in a row, reducing the likelihood that a legitimate flow is punished by accident. My belief is that *Guarded* is a considerably stronger model for these reasons, and its successes offer strong rationale to consider the best schemes for efficient information sharing. Paradoxically, *Instant* generally achieves the best performance for UDP traffic yet actively suffers when trained as a single learner—this may occur due to a roughly even spread of values between disparate actions, due to shared characteristics between legitimate and malicious flows.

Although these developments have improved upon Marl in both identified problem cases, the improvements are not quite on the order we'd expect for UDP traffic. The most likely explanation is that agents are converging to, and becoming stuck in, locally optimal (but globally sub-optimal) policies. The increased state space size makes this a more likely occurrence, as does the unclear effect of hyperparameters (α, γ) as we scale up the state space. I suspect that these difficulties may be exacerbated by the competitive nature of learning that these models embody: agents are learning action values for multiple features simultaneously, taking many actions at once (making it harder to observe the true value of each action), and controlling shared global state. Although this design does take steps to counteract such effects, these mitigations may not be enough. Moreover, benign UDP traffic shares many characteristics with attack traffic, suggesting that more training samples or some unknown feature might aid control, or that it may be worthwhile to extensively pre-train agents non-competitively on each feature using individual flows.

Most importantly, I must state that while the models and techniques presented here are a significant improvement over past RL-based work, this strand still trails behind existing (exact) DDoS flow detection mechanisms where TCP traffic is concerned. The ability to better protect VoIP traffic when compared against one of these approaches is a curious observation, which suggests that other (exact) protocol-agnostic approaches may carry hidden assumptions and is a promising direction for future investigation. Similar traffic makes up a significant fraction of network load today (18–27 %). Although this work maps the territory to some extent, there are still more advancements to be made before RL-based DDoS defence is truly competitive. The benefits at present are, however, substantial. What this offers above the approaches we discussed earlier are potentially more flexible deployments, low-overhead and fixed-cost decision-making, without requiring active measurement or the network resources and capabilities that the most effective techniques rely upon. Moreover, our decision making processes are entirely agnostic of the protocol or content of traffic, offering future-proofing against the introduction of new transports.

Security concerns and vulnerability Can an agent be flooded with new flows to reduce their ability to make decisions? One of the risks introduced by our policy update strategy is that so much work can be queued up that an agent is never able to act on some attack flows. The natural solution is to impose an upper bound on the amount of action computations/policy updates that can be performed before a work list is discarded completely. This removes the guarantee that all flows will be visited fairly often, but if updates occur regularly then this random sampling may be sufficient to achieve good performance.

Can an attack on the controller impact our approach? This question hinges upon whether the deployment environment is a traditional network or is fully SDN-enabled—each agent is, in a sense, *a* controller alongside the network's controller. In a traditional network, only the agents act as controllers, but since they periodically request per-flow data no amount of flows generates more requests or messages to the agent. More work is generated, but we discuss how to handle this safely above. Accordingly, agents can never be stalled by request volume: their only remote communication (load measurements) comes from trusted nodes, is highly periodic, and has constant size. The same logic holds for a fully software-defined network. Recalling that we do not employ the network's controller to install filtering rules on edge switches, an agent's ability to act is unimpeded. Thus, the controller is made no more vulnerable than in any other SDN. The only necessary change for such a scenario is that a load measurement which has not been updated (due to a timeout or missed deadline) should be set at $R_t = -1$.

ML algorithms have earned a reputation for eluding human interpretation, while being vulnerable to evasion and poisoning (sections 3.5 and 3.6). Given the security risks associated with introducing such techniques, it is natural to be concerned with the interpretability of the models we have proposed. With the exception of global state, the tile coding parameters we make use of ensure that the set of outputs for each feature we add is relatively enumerable: for n tilings and c tiles per dimension there are $nc^{\dim f}$ individual action value vectors per feature f (48 for the new features we introduce, 10 368 for global state), though considerably more combinations thereof ($c^{n \cdot \dim f}$). Furthermore, system state which is dependent on many signals drawn from across a wide network (such as our global state) is difficult to exert precise control over. These signals' topological separation, in concert with their burstiness and unpredictability, may have substantial effects on an attacker's capabilities.

Real-world deployment Currently, we assume that switches support an extension to OpenFlow to enable remotely installable packet-drop rules, either by running a modified version of OVS on commodity hardware at these locations or through custom firmware for egress switches. Similar functionality could be employed by making use of OpenFlow's meter rules.

Where overheads are concerned, the state space sizes guarantee that an *Instant* agent's policy remains under 520 KiB, although in practice our sparse representation typically leads to far smaller policies: ~17.8 KiB from these experiments. *Guarded* policies are 30 % of this size. As described earlier, action updates require a constant number of floating point operations. The vast majority of these operations can be vectorised trivially, if such hardware is present. Beyond this, we require that egress switches are capable of co-hosting an agent (i.e., through NFV), with the necessary hardware to support this.

Gathering and transmission of load/flow statistics would be difficult to perform as often as an emulated environment allows, without inadvertently affecting host traffic. However, the measurements acquired in such a scenario are likely to be less noisy (by being collected over longer periods of time), which could aid training. The main bottlenecks are likely in forwarding the load measurements from various aggregation points (which can be made more efficient through multicast) and in running some estimator $g(\cdot)$ to condition the reward function. We expect that agents will be able to share policies for all features, which may help to offset the reduced rate of incoming experience. Regardless, it will take longer to achieve enough state-state transitions to converge on a good policy.

One limit of SDN-capable hardware is that OpenFlow rules occupy 6 × the space of standard rules—commercial switches only have TCAM space for 2000–20 000 rules (Nguyen *et al.*, 2016). This approach consumes a rule for each active flow (the host density), and by the end of an experiment a switch can accrue around 900 rules. While we use a default fallback action to maintain connectivity, eviction of high-value decisions which filter high-bandwidth attackers poses a significant risk. Given that most flows are small, with the majority of bytes coming from a few "heavy-hitters" (Pan *et al.*, 2003), it may suffice to only apply RL-based analysis to larger flows. OpenFlow rules have an *importance*, controlling which rules may be evicted by a new entry (preventing entries from evicting those with higher importance). If an agent is to act on all flows, a solution is to assign an importance of 0 to mice flows, 1 to elephant flows, and 2 to total filtering (leaving agents to time out and remove elephant flow rules to prevent bloat). Given the high churn and prevalence of mice flows, eviction here is most likely to affect flows which are complete. In both cases, extra rules can be made available by upgrading rules which completely filter a flow into upstream blackholing as in collaborative approaches (Ramanathan *et al.*, 2018), having the agent remove this rule once blackholing is active.

4.11 Summary

Through this chapter, we have discussed reinforcement learning and how it can be used to approach the task of DDoS prevention, lending credence to

one of the claims in the initial thesis statement: 'Data-driven methods such as reinforcement learning can lead to improved performance in network optimisation and control problems, such as DDoS prevention' (*s1*). The key to doing so was to study the dynamics of the network itself—its behaviours, and realistic recreations thereof—to detect operational flaws in existing RL and algorithmic works. In turn, I designed different action models built upon a shared (and justified) model: making decisions on a per-flow or per-source basis, and relying upon learnt policies to differentiate congestion-aware and -unaware flows that methods like SPIFFY ignore.

First, we covered a large problem in modern networks: the ever-present threat of DDoS attacks—and how Internet traffic characteristics make its solution more difficult. We identified weaknesses in past remedies offered by the community, recommending instead an RL agent design which acts per flow, and have outlined the algorithmic and engineering choices needed to make its deployment feasible. Supporting this, we've examined the presented feature space in depth, offering quantitative and qualitative justification for each choice, while also expanding the global state in past Marl approaches to support arbitrary network topologies. Using simpler tile-coded policies, we have also covered how decision narrowings and per-tile updates allowed faster convergence—and independently developed methods for coalescing state which have become more common in tasks with long inference times as the field has bloomed. We have examined the *Instant* and *Guarded* action models, integrating various degrees of domain expertise with RL agents. To make real-world deployment possible in the face of obviously adversarial inputs, we have seen how it is essential to consider rate limiting (and probabilistic) strategies like TRS scheduling, and have presented a VNF- and SDN-backed system architecture. By empirical evaluation, we've seen that these new agent designs advance the state of the art in RL-based DDoS prevention, with *Guarded* agents showing the most promise for future evaluation.

While this adds another positive note to the score of DDN use cases seen throughout section 3.1, what I must stress is that this chapter emphasises the value of *co-design* and true subject-matter expertise. Networks in particular are complex, and controlled elements respond to an agent's action in ways which trace-based evaluation cannot capture—hence my disdain in the frontmatter of section 3.1. This builds on the general advice of section 3.1.6: better modelling, simulation, and understanding of the environment *led to better designs for their control.* This foundation is crucial, as it falls to us to derive the *mechanisms* of control: state and action spaces, reward functions, interaction and measurement models, and similar aspects of agent or classifier design. Learnt DDN policies work well at optimising within these constraints that we set. The final takeaway is that DDN solutions, and how we evaluate them, *must respect the complexity of the network*; evolving topologies, natural change and diversity in traffic and protocol distributions, as well as the mutation of attacks and the wider problem space.

Chapter 5

In-network Reinforcement Learning

As we have seen throughout chapter 3, DDN works—in particular, those based on RL methods—have excellent promise in the control of many aspects of the network. However, there are several consistent features in the designs of the examples seen in the literature. In order to pursue more effective policies we've seen a profusion of DRL approaches, which are computationally intense to train and execute. What this implies for the design of networks which host or apply DDN solutions is that system administrators must provision adequate compute hardware—either in commodity GPUs and CPUs, or more specialised accelerators—as well as network capacity sufficient to support the movement of operational data. These present significant sources of capital and operational expenditure, in addition to other challenges such as the space, power, and cooling requirements of such co-hosted infrastructure.

How might these hardware and deployment constraints affect the operation of any DDN system, particularly in the case of online learning? Recalling our earlier discussion on asynchronous RL (fig. 3.7), additional latency in the decision making process adversely affects both training and the effectiveness of any actions taken. This can arise from moving state and actions between their source, inference location, and final place of installation, or may originate from costlier function approximations such as larger NNs. Dedicated hosts are often required at present due to the prevalence of these more complex NNs, yet doing so incurs $\mathcal{O}$(µs) PCIe delays by moving data between the NIC and CPU/GPU (Neugebauer *et al.*, 2018; Siracusano *et al.*, 2020). Moreover, achieving reasonable model throughput such that line-rate inference can be provided requires significant batching on commodity hardware, harming median and tail latencies of any inputs. Dedicated accelerators such as *BrainWave* (Fowers *et al.*, 2018) can help somewhat here, and reduce batching (and thus tail latencies) by 32 × compared to GPU acceleration—yet inference still takes $\mathcal{O}$(ms) (Duarte *et al.*, 2019). Even

novel DMA techniques such as *GPUDirect* (NVIDIA, 2021a) halve but do not eliminate PCIe transfers.

In parallel, the recent advances we've examined in PDP hardware and the P4 ecosystem benefit us in two ways. On one hand they have produced many novel, openly available fine-grained traffic measurement techniques that can be installed in our routing infrastructure and controlled with ease. On the other, their enhanced compute capabilities have been instrumental in achieving low-latency, line-rate ML inference. From a DDN design perspective, these benefits are strongly connected; not only can we eliminate latency incurred due to batching and steering, but we can also act on per-packet or per-flow state which might be too costly to transport across the network. In this sense, PDP hardware allows us to move the entire monitoring and analysis stack (including ML inference) into the dataplane itself, and have it evolve to incorporate new approaches by changing out the set of tables and associated actions that packets must traverse. In addition, P4's control plane makes it easy to select which flows or packets are monitored in a live network[1] and potentially allows control over traffic at the decision site.

While these state-of-the-art approaches can exploit local, PDP-device-only state to offer reactive network control, the missing piece of the puzzle is learning and updating these analyses online without deferring to another machine in the network. While we have already examined how resource-constrained devices in general use bespoke data formats to make inference and learning possible (section 3.4), FPUs are excluded from the designs of all PDP device classes as they are entirely surplus to traffic processing.[2] As a result the current state of the art, as we have examined it, requires that any ML model must be completely trained offline before conversion to some PDP-friendly format, such as BNNs or a chain of MATs. While the question of data formats is well-considered, training these models online *and* in-network has not been solved from an algorithmic perspective—DNNs and their like are at odds with this goal as backpropagation is too expensive, and storage of minibatches and replay buffers runs counter to the limited memory and resources afforded to network hardware. If we can bring online learning to the dataplane, then we can take advantage of rich, local state while minimising state-action latencies as in-network ML does, while also reducing their impact on the learnt policy. This would also make it easier to train and prototype agent designs which can learn as the network environment evolves, or enable live training in testbeds and production networks when there is too little data to model and simulate a problem.

The work presented in this chapter considers how online RL can be made possible in PDP hardware, and is based upon 'Online RL in the programmable dataplane with OPaL' (K. A. Simpson & Pezaros, 2021) and 'Revisiting the Classics: Online RL in the Programmable Dataplane' (K. A. Simpson & Pezaros, 2022). Through section 5.1, I discuss and justify the data formats

[1] This is an important constraint, as state collection typically demands bytes of space in the register file per measured flow. Equivalently, if inference isn't fast enough to meet timing at a per-packet rate without pipeline stalls then this reduces performance degradation on SoC-type NICs.

[2] FPGA-based SmartNICs are an exception, where the designer may simply include their own FPU if that they have sufficient area.

which are necessary to allow both inference and learning in reasonable timescales on PDP hardware. Additionally, I also investigate how to make best use of common architectural features of SmartNIC hardware—primarily their high count of low clock-rate cores—to act and learn at low latency based on locally acquired state, without affecting packet forwarding performance. By considering efficient, parallelisable function approximation alongside these data format choices, the novel wait-free *ParSa* algorithm can then be described. This complete design is termed OPaL—On Path Learning. I then present and describe how OPaL is realised on Netronome SmartNIC hardware, including the use of platform-specific primitives and more tailored work allocation strategies (section 5.2). Section 5.3 then evaluates this implementation in depth: I investigate its throughput and latency characteristics on RL policies of varying sizes, show the resource demands of the system, and investigate performance as the compute resources allocated to OPaL are varied. Although this is a fundamentally different task from other PDP-ML tasks, I compare task execution costs against the state of the art for similarly sized inputs. Crucially, this includes an investigation of how cross-traffic carried by a co-hosted P4 dataplane are affected under various degrees of additional RL load. Having demonstrated its operational characteristics, I then describe how OPaL might integrate with state-of-the-art PDP applications to implement the RL-based DDoS prevention system described in chapter 4 rather than a VNF deployment, and comment on how operators may make best use of different OPaL agents within their networks (section 5.5). Finally, I summarise the findings of this chapter in section 5.6.

5.1 Design

Based on the design principles and problems outlined above, I present my design for an in-NIC, task-independent, online RL system—*OPaL (On Path Learning)*. At a high level, OPaL is designed to use the auxiliary compute exposed by general SmartNIC devices to offer low-latency online learning, scaling according to available on-chip resources at build time. Its design is based on meeting the following constraints:

Low state-action latency. RL DDN applications incur $\mathcal{O}$ (ms) latencies due to a combination of expensive function approximation, batching, state steering, and PCIe handoffs. Ideally, to keep pace with packet rates upwards of 40 Gbit/s we require inference or state-action latencies around $\mathcal{O}$ (ns) or $\mathcal{O}$ (µs). This would enable fine-grained operation on traffic, particularly latency-sensitive control problems. Where possible, this should correspond to increased throughput to better enable the processing of line-rate traffic.

Effectively employ parallelism. As discussed in chapter 2, SmartNIC devices often contain large numbers of slower cores without FPUs. As such,

to achieve strong latency or throughput bounds we must employ and design algorithms which are both computationally cheap (forbidding DNN backpropagation) and parallelisable. Crucially, this also allows users to scale up or down their resource costs to dedicate as many or few cores as needed to meet desired latency or throughput targets.

Use on-device state without stalling packets. In spite of the above performance goals, at larger policy sizes it becomes more likely that the inference and update steps of an RL algorithm will violate packet or pipeline timing constraints. However, we need access to state from the data-plane to meet our goal of low-latency processing. Thus, an in-NIC RL agent must interact with but not execute on the main packet path.

Reconfigurable. To simplify deployment as network operators' needs change, an OPaL RL agent must be able to be easily repurposed at runtime—i.e., without recompilation and installation of firmware or FPGA designs. While this includes complete model changes, the most useful aspect would be the ability to swap between online and offline modes of operation to increase decision throughput. Moreover, we aim to make use of the control-plane for easier selection of target flows.

Minimise resource use. Due to the resource-constrained nature of PDP hardware, applications and packet pipelines have highly limited TCAM-accelerated or other high-speed memory (e.g., $\mathcal{O}$ (KiB)). As such, storing replay buffers is infeasible, as are RL algorithms which require such buffers for stable learning. This is amplified when learning a shared policy from several flows concurrently.[3]

[3] This consideration has historically been labelled as parallel RL (Grounds & Kudenko, 2007), not to be confused with the multicore/parallel algorithms we are also interested in.

To meet these constraints, OPaL operates alongside a co-hosted P4 data-plane in a SmartNIC, running asynchronously with respect to the packet forwarding path, with its full interaction model given in section 5.1.1. To achieve both inference and learning I employ fixed-point Q numbers as the main data format, and justify this against data formats in other embedded environments (section 5.1.2). Classical RL algorithms and function approximation schemes—semi-gradient Sarsa and tile coding (sections 3.2.1 and 3.3.3)—enable learning under the memory and computational constraints of PDP hardware. This considers different parallel processing strategies, as well as the conversion to a wait-free algorithm enabled by using Q numbers as the primary data format (section 5.1.3).

Bringing OPaL to PDP *switches* such as the Tofino would be difficult, if not impossible as they closely match the P4 PSA, leaving no spare general-purpose compute units. Taking inspiration from real-time programming, a potential solution might be to divide RL processing across several packets (i.e., computing a portion of the preference list each time) until further work would delay outbound transmission. This would introduce new issues in concurrent access, work splitting, and altered timescales for learning: we leave their treatment to future work.

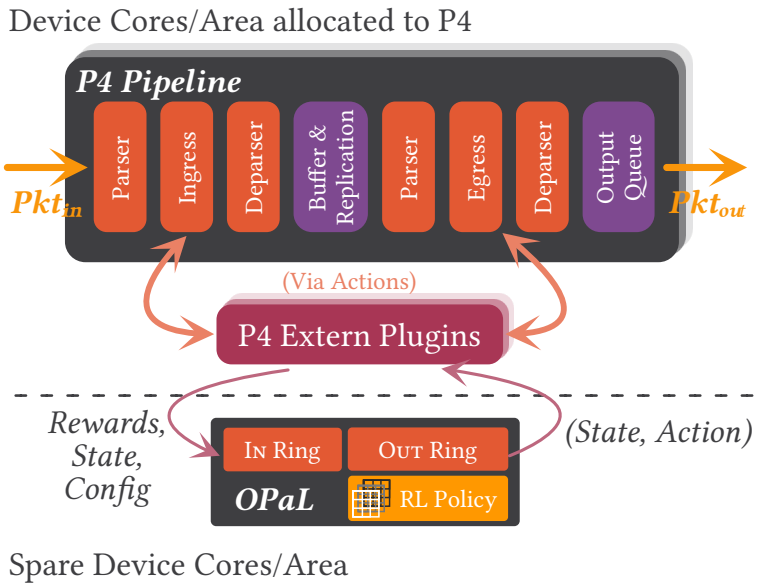


Figure 5.1: SmartNIC-type PDP devices typically implement a target dataplane design by mapping the processing pipeline across one or more cores or FUs. Such devices are well-suited to this; NPUs or SoCs such as NFPs often have spare cores which aren't dedicated to running the dataplane, while spare area on FPGAs may be used to design arbitrary FUs. While additional compute resources can relax the timing constraints needed to hit line-rate throughput, to prevent packet stalls as required we must move long-running computations (i.e., RL inference and updates) to this auxiliary compute. P4 programs expose device-specific functionality through `externs`, allowing any extra compute resources and *Inter-Process Communication* (IPC) to be accessed from ingress and egress tables, thus providing asynchronous execution and fast transmission of P4-extracted state as needed.

5.1.1 Interaction and system model

OPaL is a general, task-independent framework for in-network, online training and execution of *any reinforcement learning agent design* using classical methods. OPaL is agnostic to the meaning of state vectors it receives as inputs and the actions it produces, which are employed by other functional units or the dataplane. However, in-NIC or in-network execution specifically benefits packet-, flow-, and network-level learning, control, and optimisation tasks. OPaL meets the above goals by operating as a system which exists *in parallel* to a co-hosted P4 dataplane. Figure 5.1 describes and demonstrates the requisite interaction model; state extraction (i.e., flow telemetry collection and processing) occurs in the ingress and egress MATs of the P4 dataplane. The packet pipeline of a P4 dataplane then communicates with OPaL via `extern` plugins using In (state, configuration) messages to reconfigure the agent or request inference, and Out (action) messages which carry output state-action pairs for the environment to make use of. The contents of these messages are explained below. To deploy this design, we require a platform-specific implementation of OPaL itself and the In/Out

ring interaction mechanisms—exploiting how SmartNIC devices often expose general-purpose compute. As many of these devices have engineering and development histories which predate P4, such general compute beyond the P4 PSA's limits (The P4.org Architecture Working Group, 2021) is surprisingly common. This then provides path-adjacent, on-chip RL in the dataplane.

OPaL itself runs on one or more cores of a SmartNIC to convert state measurements of a known size from the environment into a stream of actions using a stored policy. I discuss how it scales with additional compute as part of section 5.1.3. These dedicated cores are then responsible for processing requests, computing actions, and updating the underlying policy in real time. Combined with reward measurements, this policy can then be updated or trained from scratch entirely on the NIC, acting as a fully online RL agent. An input state vector *always* induces an action and, if online learning is desired, updates the policy using either an included reward or one retrieved from memory according to a key placed alongside the state. This allows for simultaneous control and learning over independent systems by the same agent (i.e., optimising several flows with their own reward measures, such as DDoS mitigation in an AS where each next-hop AS might have their own health metric). Configuration messages may be provided over either the data or control-planes, where the P4 control plane may be used to provide access control over which machines or ports may send such commands.

By executing on spare compute units, this design prevents packet stalling by moving longer-running compute out of the packet path, and can scale with cores made available at compile-time to improve latency and throughput. By operating as closely as possible to the P4 pipeline, OPaL uses and learns from per-packet state with minimal added latency (avoiding PCIe transfers and batching as required), while imposing minimal impact on carried traffic for both bump-in-the-wire deployments and at end-points. Moreover, the presence of the P4 dataplane allows easier inclusion of existing P4 traffic measurement and state extraction techniques, such as those covered in chapter 2.

Execution trace handling. To be generally applicable, we require that OPaL is flexible in how reward values are mapped to input state vectors. Target applications may aim to control one or many separate trajectories—i.e., learning from concurrent traces (Grounds & Kudenko, 2007)—and in the online case we must have a low-overhead method of retrieving the correct last state-action-value tuple. Moreover, each may require its own reward value depending on the nature of the MDP, for instance when optimising individual flow behaviour rather than joint control to benefit a shared environment. OPaL thus allows several sources for selecting these values, which may be configured separately for trace and reward selection:

Shared. A single RL trajectory or reward value is held. The reward value must be periodically updated using dedicated reward packets.

Field. A given field of the input state vector is used as the lookup key (e.g., in a hash table) for either the trajectory or reward value. For instance, this may be a packet's flow hash or source IP. The reward value must be periodically updated as above.

Raw Field. A given field of the input state vector is used as in *Field*, but is not hashed for lookup.

Value. A given field of the input state vector is directly used as the reward value. The installed policy can use or ignore this field as needed.

Policy constraints. To minimise (and predetermine) the amount of data required to encode a policy, as well as reduce the computational complexity of policy inference, tile coded policies are constrained in the following ways. All tilings are assumed to be uniform rectilinear grids as in algorithm 1, where each dimension of input state has a single minimum and maximum assigned. We assume that each dimension is subdivided into the same number of tiles, and that all tiling sets contain the same number of overlapping tilings over a given list of dimensions. Each tiling set then covers a list of dimensions up to a platform-defined size.

Message formats. To provide the needed functionality and runtime configuration, OPaL's In ring receives the following message types.

State messages. These contain only a list of input data values from the environment, which invokes an inference and/or update cycle.

Reward messages. These contain a single data value, and an optional data value used as a key to ensure it is mapped to the correct state trajectory.

Configuration. These messages contain either top-level parameters (action counts, dimension counts, hyperparameters), or the dimension list for each tiling set.

Policy insertion. As network packets have a restricted size, these messages subdivide a full policy parameter vector, and contain an array of tile value data alongside an *offset* into the policy. These may be used to insert a pretrained policy into an offline agent, or to percolate policy updates among online learning agents in a network.

OPaL's Out ring only carries a single message class, which is a state-action tuple.

5.1.2 Data format

To implement online learning such as in RL, we require a data type which allows us to perform numerical computation without an FPU—principally, to compute the values in action preference lists. Moreover, to achieve online learning we require a format which is both fast to work with (to minimise processing latencies), and suited to represent gradients and temporal difference values, i.e., incremental changes to stored policy weights. Recalling the discussion in section 3.4, we employ *fixed-point arithmetic*; it offers both the versatility needed to be easily reconfigurable, and it maps simply to ALU operations. For instance, only multiplications and additions require additional bitshifts for base pre-/post-conversion, while additions and subtractions require no additional overhead.[4]

[4] Depending on the system design, (hyper-)parameters used only in divisions can be replaced with right bitshifts if we restrict their allowed values to negative powers of 2. This comes at the cost of system flexibility, and as such OPaL's implementation doesn't make use of this possibility.

From a configurability perspective, the count of fractional bits in a Q number can be easily changed at runtime. Naturally, this has no effect on overall latency and throughput of fixed-point arithmetic, but is a useful characteristic for being able to deploy different RL agents to the same hardware without invoking more costly firmware or FPGA design installation. If this is fixed at the same setting for all values used, then we needn't tag individual Q numbers with information about their base, saving memory. The bit width of the numbers themselves (i.e., k) may be changed only at compile time in many cases, particularly as SmartNICs often lack dynamic memory management in their native non-P4 programming environments. Lower choices of k sacrifice numeric range, but allow policies and data to occupy less memory—allowing fairer resource use versus other dataplane programs—and thus occupy less memory bandwidth in data transfers. Alternatively, larger policies may be stored in the same memory bounds (potentially enabling the solution of more complex problems).

The reduced width and precision floating-point data formats we also examined earlier are not suitable here, in spite of their successes in other embedded domains. First and foremost, these still require specialised FPU implementations. This is naturally at odds with the design and goals of PDP hardware, and beyond FPGA designs such a data format would be infeasible. Software floating-point emulation has historically required 10–30 × more cycles to perform compared to hardware FPUs (Iordache & Tang, 2003), which is incompatible with our need for low-latency and high-throughput processing. While the added dynamic range would be useful in this application, performance is a primary goal.

PDP hardware excels in applying actions to network packets using MATs, potentially giving us a high-performance method to install selected actions within the network. However, directly modifying these match tables from within the device itself is neither feasible nor safe. On Netronome NFP hardware in particular, rule updates *must* be applied by the co-hosted controller machine, as tables are reliant on the optimised DCFL (Taylor & Turner,

2005) data format. In addition to the prohibitive complexity of building this data structure on-device, its construction requires knowledge of the entire rule set (and cannot be incrementally updated). I instead suggest in general that `extern`s or datapath stages which apply RL actions to packets should maintain a small store of state-action pairs, and periodically send these back to the controller for batch installation. This ensures that the majority of installed rules benefit from faster hardware-accelerated lookups, while preventing installation delay on the newest decisions. Platforms such as Intel Tofino greatly simplify this, where Tofino Native Architecture intrinsics such as *Action Profiles/Selects* allow a P4 action to be chosen based on a register value (e.g., an RL action). Future NICs and SmartNICs may expose support for runtime table modification via the Portable NIC Architecture (The P4 Language Consortium, 2021) as discussed in section 2.2.4, but at present these proposals are under constant revision and are far too nascent to seriously consider.

5.1.3 Algorithm

To enable online in-NIC learning in spite of the computational limits of PDP hardware, we must return to *classical* RL methods and models. In particular, we focus on tile coding with one-step temporal-difference learning algorithms such as Sarsa, which were discussed and explained earlier (section 3.2.1 and section 3.3.3). These functions do not require batches of inputs to learn in a stable way, negating the memory needed to store experience replays, and have simple update and inference logic. Moreover, gradient computation is identical to the forward pass and has no dependency on the current parameter values $\boldsymbol{\theta}$, potentially allowing hit tiles to be stored to accelerate the next update step. Finally, the choice of single-step algorithms (as opposed to n-step or Monte Carlo methods) bounds the amount of per-trace state required for online learning to just the last state-action pair, safeguarding the limited memory of our target devices.

As for how to take advantage of the multicore nature of SmartNICs, there are two strategies worth considering. Suppose we have n cores. The first strategy is that we simply have each core work independently. For instance, when dealing with a stream of input state vectors according to OPaL's interaction model, we may serve each arrived vector to a free core—this core then tile codes the state against every tiling, produces an output action, and then updates the policy as required (algorithm 1 and eq. (3.6)). Intuitively, this produces an $n\times$ throughput improvement assuming there are no bottlenecks at either the input and output queues or mutual exclusion around shared data. However, this offers no reduction to the processing time of any *individual* element—and so the state-action latency is not reduced as desired.

To attain the latency improvements we desire, we must consider instead a

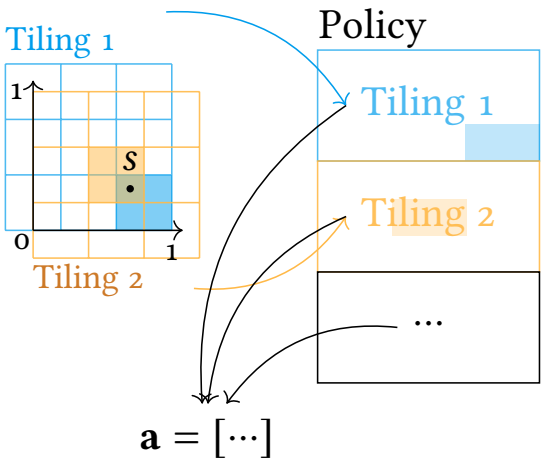


Figure 5.2: Tile-coded function approximation can be considered as a map-reduce problem during both the inference and update steps. Action preferences are aggregated from *disjoint* tile queries, where each tile hit contains a list of action values to add to $\mathbf{a}$. Retrieving or updating each tiling is thus a separate task. As all constituent action preference lists are disjoint between tasks, updating hit tiles requires no control over concurrent accesses—furthermore, it requires no final aggregation step.

second strategy: how several cores may work together to complete an individual inference or update operation. Consider that tile coding computes individual hits against a set of tilings to produce a sparse boolean feature vector, which we'll denote $\mathbf{x}$, and refer to the set of all its non-zero indices by H. Without loss of generality, suppose we have two tilings t_1 and t_2, with a number of tiles $|t_1|$ and $|t_2|$ in each respectively (such that $\dim \mathbf{x} = |t_1| + |t_2|$). For notational simplicity, let each entry of an agent's policy data $\boldsymbol{\theta}$ be a vector of length A (i.e., choosing between A discrete actions). A state vector will produce exactly one hit in each tiling, having indices $h_1 \in [0, |t_1|)$ and $h_2 \in [|t_1|, |t_1| + |t_2|)$, without overlap. The final action preference list is given by eq. (3.1), which we can specialise further:

$$\mathbf{a} = \boldsymbol{\theta}^\top \mathbf{x} = \boldsymbol{\theta}[h_1] + \boldsymbol{\theta}[h_2] = \sum_{h \in H} \boldsymbol{\theta}[h]$$

As a result, tile coded RL inference may be subdivided into several distinct computations of tile indices whose results are added together as a final aggregation step. Most of the work in each task arises from computing the tile index hit by the state vector. Crucially, we have shown that memory regions of each tiling have no overlap with one another by construction. Each memory address is visited and owned by exactly one task—hence, there is no concurrent access to policy values—and so no locks are required to protect each region of the policy. Figure 5.2 demonstrates this process visually.

When updating the policy we need only monitor the number of completed tasks rather than perform a final aggregation step. To see this, we specialise Sarsa's update step (eq. (3.6b)) using our list of hit indices H. After centrally

computing a TD value δ_t using $\mathbf{x}$ and a subsequent $\mathbf{x}'$, the action a invoked by $\mathbf{x}$, and a learning rate α, we update the policy values involved in the previous decision (using Python slice notation for simplicity):

$$\begin{aligned} \boldsymbol{\theta}[:, a] &:= \boldsymbol{\theta}[:, a] + \alpha\delta_t\mathbf{x} \\ \therefore \boldsymbol{\theta}[h_1, a] &:= \boldsymbol{\theta}[h_1, a] + \alpha\delta_t, \\ \therefore \boldsymbol{\theta}[h_2, a] &:= \boldsymbol{\theta}[h_2, a] + \alpha\delta_t. \end{aligned}$$

and so on for all $h \in H$ when we have an arbitrary number of tilings. Thus, using the above knowledge that concurrent accesses to any parts of the policy cannot occur, an RL update divides into subtasks without locking in much the same fashion as inference does.

The aggregation step still risks becoming a bottleneck during inference; if individual preference lists are passed back as discrete messages, then this requires explicitly iterating over and summing all such lists. While this may be performed by a dedicated worker thread or core in parallel to the other disaggregated inference tasks, this involves additional costs in every such task for `memcpy` operations into a message buffer. Consider again that OPaL targets PDP hardware: without a dynamic memory allocator, this buffer has bounded length and so risks causing head-of-line blocking for all other tasks. However, encoding our values using a fixed-point representation allows us to employ atomic instructions to remove the aggregate step, thus producing a wait-free algorithm. Recall that the aggregation step involves *only additions*, that atomic integer fetch-add instructions are commonly offered on many machine architectures,[5] and that fixed-point addition is identical to integer addition. As a result, the final action preference list may be allocated once and atomically added to by all workers. Moreover, if extra care is required to prevent numeric overflows or ensure saturation, then any worker may locally verify whether the fetched value and summand would cause an overflow. Alternately, an implementer may simply enforce upper and lower bounds on each action value to prevent positive or negative overflow.

[5] While other datatypes such as `f32` can technically support atomic operations on many platforms, outside of specialised GPU environments such as CUDA these incur non-trivial performance costs due to cache and register placement restrictions.

Parallelising the RL algorithm among processes in this way thus requires tight coupling between the function approximation and RL update algorithm. The combination of all of these elements forms the basis of *ParSa*—parallel Sarsa (algorithm 2). To match the deployment environment of SmartNIC devices, ParSa is presented such that each worker thread operates in an infinite loop to await and process requests delivered over a message channel In, and produce a stream of outputs on Out. We first assume that every process knows its own index—*id*—and that a *schedule* has been precomputed to divide individual tilings among cores as tasks (line 5). This careful division is important; an individual tiling may contain several dimensions, and by recalling algorithm 1 it is obvious that tilings with a higher dimension count require more iterations and are thus more expensive to infer. To give some context on the number of tasks ParSa is expected to perform, a policy sized to match the agent designs from chapter 4 with one bias tile and 16 sets of

Algorithm 2: ParSa—*Par*allel *Sa*rsa

```
/* Given message passing mechanisms scatter and recv, an
   input request stream In, an output action stream Out,
   quantised arithmetic functions Q_mul and TileCode, and
   omitting schedule/config/precache/reward updates.      */
/* cfg.α, cfg.γ are hyperparameters affecting the
   significance of each update and the degree of
   forward-planning, respectively.                        */
1  enum Par { Act(state), Upd(delta, action, state) };
2  const cfg, policy = /* ... */;
3  let values: [AtomicI32; cfg.n_actions] = {0};
4  let acks: AtomicI32 = 0;
5  Function ParSa id, schedule
6      if id==0 then
7          forall state_pkt in IN do
8              Ctl(state_pkt);
9      else
10         while true do
11             Minion(schedule[id − 1], recv());
12 Function Ctl state
13     values, acks = {0}, scatter(Par::Act(state));
14     acquire slot for OUT, copy state into slot;
15     await acks == cfg.n_minions;
16     let action = argmax(values);
17     write action into OUT slot, enqueue;
18     if cfg.online then
19         let ((l_state, l_act, l_val), found_s) =
             cfg.lookup_state_from_key(state);
20         let (reward, found_r) = cfg.lookup_reward_from_key(state);
21         if found_s && found_r then
22             let δ_t = reward + Q_mul(cfg.γ, values[action]) − l_val;
23             δ_t = Q_mul(cfg.α, δ_t);
24             acks = 0, scatter(Par::Upd(δ_t, l_act, l_state));
25             await acks == cfg.n_minions;
26         cfg.store_state(state, action, values[action]);
27 Function Minion tasks, msg
28     switch msg do
29         case Par::Act(s) do
30             forall task in tasks do
31                 let hit = TileCode(s, task);
32                 for i in [0..cfg.n_actions) do
33                     values[i].atomic_add(policy[hit][i]);
34         case Par::Upd(δ, a, s) do
35             forall task in tasks do
36                 let hit = TileCode(s, task);
37                 policy[hit][a] += δ;
38     acks.atomic_add(1);
```

8 tilings creates 129 work items. As such, a schedule is required to produce the most balanced workload between worker threads.[6] However, this is a deployment-specific implementation detail, as in addition to raw task costs an effective scheduler must account for cache size limits and physical/logical core differences. I present an allocator tailored to Netronome hardware later in algorithm 3. A single worker thread, designated as the *controller*, then receives and processes state vectors from the environment (lines 6–8), while the remaining threads—*minions*—retrieve a broadcast message of type *Par* to apply over their prescheduled task set (lines 9–11).

[6] Work stealing or other parallelisation strategies are not suitable choices for work balancing in tile coded Sarsa for several reasons. These include the computational simplicity of individual tasks, the predictable compute cost of any such task, and the relative cost of IPC required to pass cached state between workers.

The *Ctl* procedure directs tasks to minion threads, manages storage of execution traces, and computes the policy adjustments prescribed by the Sarsa TD value. Applying the insights of Travnik *et al.* (2018) to minimise action latency, an action is computed and sent out into the environment—and only then is the underlying policy updated. For each input state, the controller thread zeroes out the shared aggregation space, where *values* holds the aggregated action preference list and *acks* contains the count of terminated minion threads. An *Act* request is then sent as a broadcast message to all minion threads using the platform-specific *scatter* IPC call (line 13). While the minions produce the action preference list asynchronously, the control thread then requests an output message slot and prepares it by copying the state into the acquired buffer (line 14).[7] Once every thread has marked its termination, the values of each action given the input state are located in *values* (line 15). The index of the largest value is then taken to be the chosen action, though this may be trivially modified to account for ϵ-greedy selection (line 16), which is then placed into the output message and returned to the environment (line 17). When online learning is enabled, the control thread checks for a reward signal and prior state-action-value tuple according to the configured trace selection scheme from section 5.1.1 (lines 19–20). If a match is found, it computes and reduces by α the Sarsa TD value δ_t (lines 22–23), before passing this adjustment and the prior state action pair to the minion threads to perform the update step (lines 24–25). Modification to other single-step algorithms such as *Q-learning* would be simple, requiring only changes to the computation of δ_t. The newest state-action-value tuple is then stored (line 26).

[7] In principle, the controller thread may also act as another minion after this step, between scatter and recv calls, subject to code store limits.

The *Minion* procedure processes a single request to retrieve and aggregate, or update, the action values learnt for an input state over some subset of the policy tilings, *tasks*. All minion threads cover the complete set of tilings between themselves. In the case that action inference is requested, the policy hit index is computed against the input state for each tiling in this thread's task list (line 31). As described above, each action value is atomically added to its matching position in the shared preference list *values* (lines 32–33). When an update is requested, all tile hit indices are recomputed (line 36), and an adjustment δ is added to the selected action in each case (line 37). Once all subtasks have been completed, this thread's completion is recorded (line 38). While it is apparent that the indices of hit tiles

may also be locklessly sent back and stored as part of the execution trace—i.e., by adding an extra value slot per task—the number of subtasks often far exceeds the count of values in a state vector. As a result, this results in a larger `memcpy` in *Ctl* (the serial portion of ParSa), and as such it is more efficient to recompute hit indices in the update step—though this optimisation is useful in the case of the first (non-ParSa) parallelisation strategy.

As required, this new algorithm scales with increasing processor counts to reduce both the inference and update costs incurred by a single state vector. This lowers the state-action latency and produces a corresponding increase to throughput. However, the factor of speedup is *worse* than $n\ \times$, and exhibits diminishing returns in terms of core count—for instance due to the cost of `memcpy`s and arithmetic in the serial logic of *Ctl*, message passing costs at the broadcast and receive steps, and any memory fences or reorderings inserted by the compiler around atomic arithmetic. This is limited further by the number of subtasks which can be generated for a given policy. While ParSa assumes tile-level granularity, it may be modified to compute individual actions at the cost of being dominated by significant repetition of work and IPC overheads. Since these aspects are platform-dependent, this behaviour can only be shown empirically as in section 5.4.2.

5.2 Implementation

OPaL is implemented as a collection of programs targeting the Netronome NFP family of SmartNICs, using a mixture of the proprietary Micro-C language and P4. As this work's implementation relies upon a good amount of platform-specific intrinsics and optimisations, it is necessary to explain some of the NFP's basic architectural details. These SoC devices achieve scalable packet processing through sheer parallelism. Most of the chip is composed of MEs—physical cores—grouped into *islands* of 4 or 12 MEs. All 12-ME islands are used by a default P4 pipeline, while two of the 4-ME islands are left free for user code. Each ME has 4–8 *contexts* (hardware threads) which share a code store. Beyond registers, the platform implements an explicit memory hierarchy scaling in size, location, and access cost:

$$\text{LMEM (ME)} < \text{CLS (Island)} < \text{CTM} < \text{IMEM (Chip)} < \text{EMEM}$$

Interested readers will find more in-depth detail presented in appendix C.

This section covers how policy data is stored in OPaL to make best use of the above memory hierarchy (section 5.2.1), and how cores and other resources are applied to implement both parallelism strategies described above (section 5.2.2). I further detail how OPaL implements the In and Out rings on NFP hardware (section 5.2.3). Through section 5.2.4, I discuss and introduce the design of efficient work-passing and aggregation mechanisms on NFP hardware required to enable the *ParSa* algorithm. Section 5.2.5 describes

how OPaL may be configured at compile time and during runtime. Finally, I detail the scheduler used to partition tile coding tasks between available worker threads in section 5.2.6.

5.2.1 Policy storage

Taking advantage of the non-uniform memory architecture in NFP hardware, OPaL splits its policy across the CLS, CTM and IMEM memory regions. This arises both from necessity, and in the pursuit of runtime performance.

Firstly, policy data is stored densely, as the nature of embedded programming means that all required memory must be statically allocated. As such, the program must reserve enough memory at compile time to contain *any policy*. While this does not rule out sparse storage, this would introduce a lookup overhead when finding the memory address of each tile's data (as well as at-rest storage costs for, e.g., a hash table). The price in memory has already been paid for storage of a full policy, and so a dense strategy simplifies lookup by assigning a fixed (and easily computed) array index to each tiling set.

Secondly, recalling that a worker must retrieve an action preference list for each tile, we aim to minimise the latency of each memory access as part of our overall performance goal. Ideally, this would mean placing the entirety of the policy into CLS, but there is insufficient space to do so.[8] Suppose we've chosen $k = 32$ for our fixed-point format, thus each action value occupies 4 B. Given a action values, d dimensions in a tiling, s tilings in a set, and t tiles per dimension, that tiling then occupies $4asd^t$ B of memory—its cost scaling exponentially with the count of input dimensions. Consider the *Instant* agent design of chapter 4: the largest tiling set chooses $a = 10$, $d = 4$, $s = 8$ and $t = 6$, and thus requires 1280 KiB. This alone exceeds the 64 KiB of CLS available per island. Worse still, we must include enough space to store *any* tiling set below the maximum parameters. If we do not differentiate between tiling sets according to dimension count, then for *Instant*'s 17 tilings we would require 21.25 MiB.

[8] CLS is the lowest-latency memory region which is practically usable: LMEM is dedicated mostly to program state and variables which spill from the register file, and cannot be accessed across ME boundaries.

The solution then is to explicitly partition the policy's storage across these memory regions according to maximum dimension count; i.e., assigning tiling sets having $d \leq 1$ to CLS, $d \leq 2$ to CTM, and $d \leq 4$ to IMEM. As we cannot fit a whole multidimensional policy into CLS, this design instead maximises the proportion of the policy which is placed into smaller regions. The above dimension limits are arbitrarily picked to match *Instant*/*Guarded* as before, but can be customised at compile time subject to resource limits. This increases the proportion of accesses made to lower-latency memory. Moreover, these memory regions are accessible to all MEs on the same island, and with increased access cost for remote islands.

Parallels in other platforms This tiered division of policy regions applies to other device classes as well. For instance, FPGA devices have a similar hierarchy between LUTs, block RAM, SRAM and *Dynamic Random-Access Memory* (DRAM).

5.2.2 Action and update computation

I implement both parallelism strategies described in section 5.1.3, each as a separate firmware model governing how the compute-heavy parts of these tasks (action selection, policy updates) are carried out:

Ind ***(fig. 5.3)*** Separate threads listen for new states, and each performs its work sequentially. Computing an action list requires a *read lock* on the policy. If an update occurs, the core requests a *write lock* before updating, greatly limiting online throughput. *Tile lists* are stored in each state trajectory for update computation.

CoOp ***(fig. 5.4 and algorithm 2)*** Threads cooperate on processing state vectors, minimising latency. *Minion* threads have a fixed list of work items, while a *controller* thread sends compute and update commands before awaiting worker completion. Work items are disjoint, requiring no policy locks. *State vectors* are stored in each state trajectory for update computation.

Both designs interact with the environment using the *Multi-Producer/Multi-Consumer* (MPMC) channels described in section 5.2.3. *CoOp* also employs carefully optimised communication between workers and runtime enumeration (section 5.2.4).

Each offers a different point of optimisation; if updates are disabled, then the *Ind* model can maximise throughput, while the *CoOp* model is designed to minimise decision latency and needs no locks to update the policy (increasing *online learning* throughput). These correspond to only executing a trained policy and actively (re-)training a policy, respectively. Latency and throughput, as in many networked systems, have different effects upon RL agents according to their design and target problem. Higher RL throughput is a necessity for per-flow or per-packet applications, which can require high decision-per-second rates even after combining state measurements received from the environment, such as flow control in DDoS prevention. Equally, lower latency affords an agent finer-grained control and learning of a problem, being able to react sooner to new information (e.g., device state in a routing optimisation problem, or queue depth when trying to enforce packet pacing).

In both cases, the configuration data structure holds a cache of adjusted minima, maxima, tile widths, and shift amounts for each tiling. As this data resides in nearby CLS, this offers a reliable way to accelerate inference and

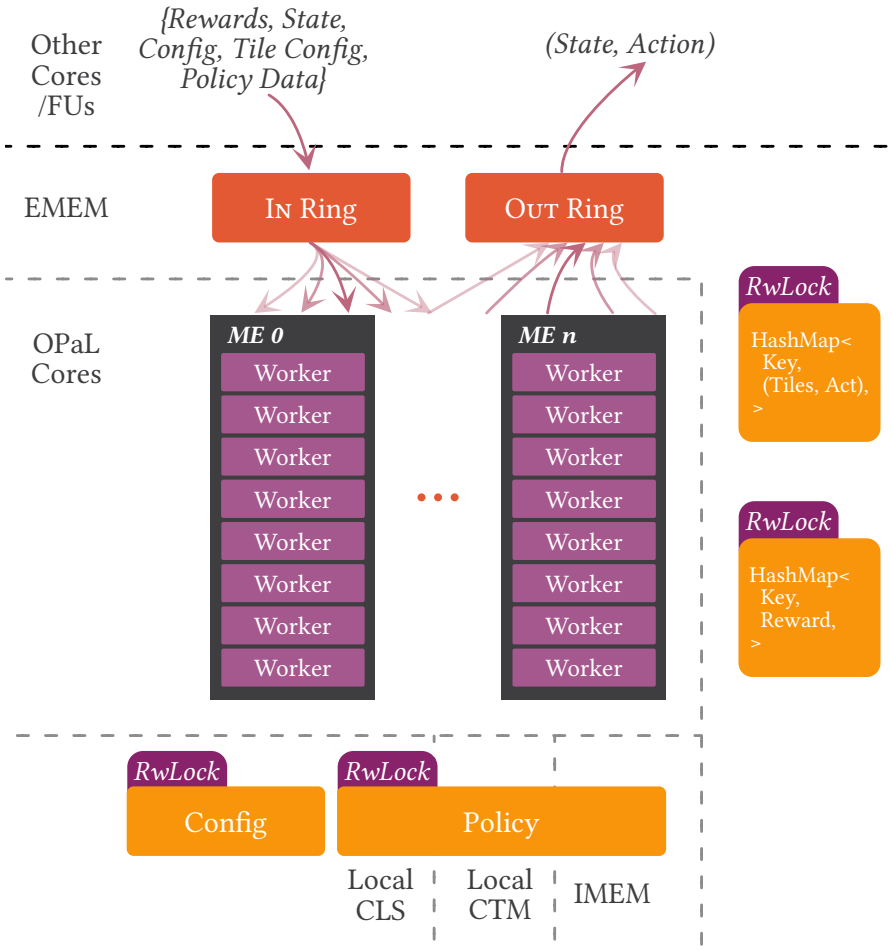


Figure 5.3: The *Ind* firmware design implements the first parallelism strategy discussed in section 5.1.3, where each thread *independently* performs tile coded inference and Sarsa RL updates. *Workers* each pull commands from (and push actions to) the environment over the In and Out channels. To maintain consistency between all workers, configuration, policy data, and state-reward trajectory data must be guarded by read-write locks. During inference, workers acquire a shared read lock around, e.g., policy data. As a result, *Ind* optimises throughput for an *offline* agent, but because policy updates require an exclusive write lock around many parameters, only a single worker may perform an RL update at any time.

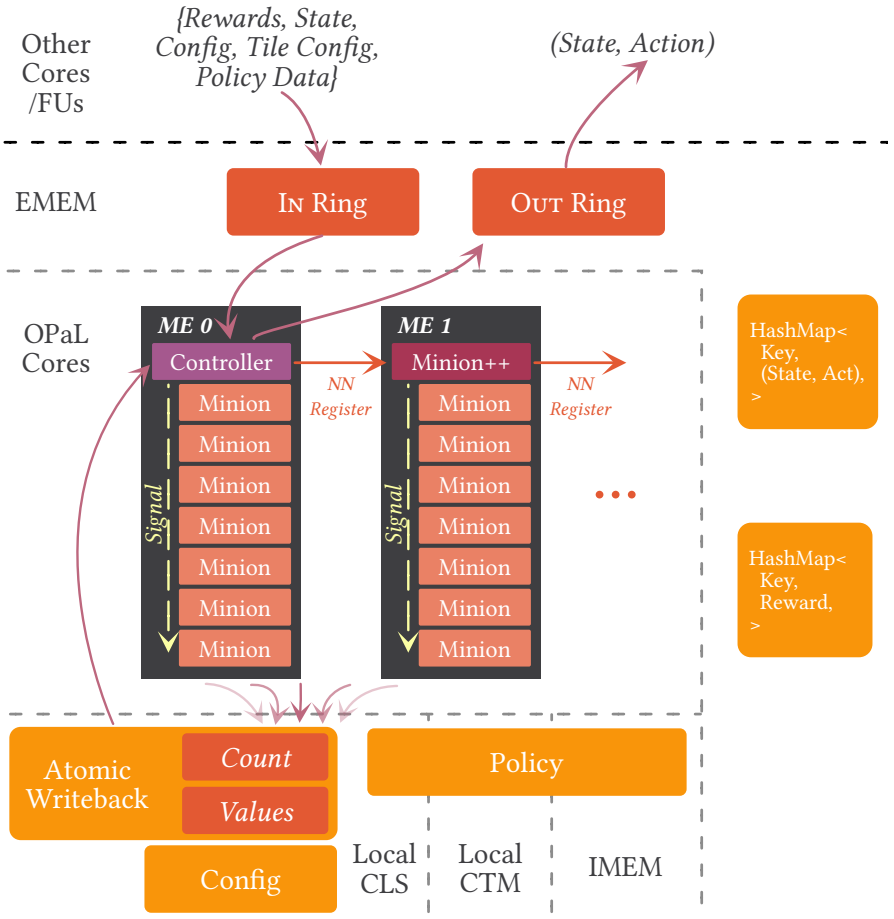


Figure 5.4: The *CoOp* firmware design implements the second parallelism strategy discussed in section 5.1.3 (ParSa, algorithm 2), where each thread *cooperates* on tile coded inference and RL policy updates. A single *controller* thread interfaces with the environment over the In and Out channels, and then delegates RL computation and updates to many *minion* threads, who operate on independent subtasks. These messages are moved between MEs using specialised next-neighbour registers. The first context of each physical core is responsible for placing this message into shared scratch, notifying the other contexts on its ME, and forwarding the message to remaining MEs on the island. This design minimises state-action latency, but crucially maximises *online* throughput by having no mutually exclusive data access.

updates in all OPaL threads. Once a *CoOp* minion receives its task set, it pre-computes each task's memory tiling tier, the index of that task's tiling set, and internal index in the set of that same task, accelerating lookup of these parameters from task indices. *Ind*, when learning online, additionally caches the full hit tile list as part of the execution trace, as its cost is far smaller than inference—in *CoOp*, it is paradoxically cheaper for each worker to simply repeat the tile coding over its task subset due to the additional serial work described in section 5.1.3.

Although not covered directly in algorithm 2, adaptation to support ϵ-greedy action selection requires an additional consideration for slowly-annealed epsilon values. If ϵ is reduced by an amount smaller in magnitude than the current number of fractional bits can represent, then we instead reduce ϵ by the smallest valid amount every T decisions.

5.2.3 Agent-environment communication

OPaL uses MPMC messaging channels to communicate with other elements; be they P4 programs on the packet path, or other on-chip analysis and control modules. This allows decisions to be made asynchronously—preventing packet stalling—and allows many RL agents to be used if desired. The key insight of this mechanism is that on-chip reward and state signals enjoy first-class support in the same manner as packets from the P4 dataplane, allowing agents to act on environmental signals from other on-NIC and -chip asynchronous processes, or the controller itself. As such, OPaL can receive input from P4 `externs` or other, dedicated off-path flow state measurement applications.

This implementation uses platform-specific IPC—*EMEM ring buffers*—as the basis for MPMC communication over the In and Out rings. These are NFP-intrinsic primitives which allow an arbitrary number of listener threads to await the arrival of any work item using hardware signalling. While this signalling and delivery is specially hardware-accelerated, this comes at a cost of strict message body size limits; unfortunately this falls short of the maximum state-action pair size, let alone arbitrary policy packet payloads. To work around this, OPaL maintains a freelist of byte slices for both the In and Out channels, while EMEM ring messages themselves carry lengths and pointers from the freelist alongside any preliminary P4 parser data. As PDP hardware lacks dynamic memory allocation, large buffers are allocated at compile time with many fixed-size slices; In's buffer slots are sized to hold MTU-size packets, while Out's slots hold enough bytes to store the largest possible state-action pair. The OPaL controller with the lowest index locks each of these lists and populates it using all contained slices, from which point any other thread in the SmartNIC may lock the structure to request or return a valid message pointer. This costs a median 126–140 ns communication time for pointer-sized (4 B) messages depending on the locality of the

Table 5.1: Median IPC messaging costs on NFP hardware for 4 B payloads, measured over 65 536 trials. Of these, only EMEM rings can be used between islands, while nearest neighbour registers have strict placement and access constraints. These one-way delays are measured by halving the RTT between two cores, or subtracting a return reflector write cost for next-neighbour registers due to their one-way access limits.

IPC mechanism	Cross-island support	Cycles	Time (ns)
Next-neighbour registers	✗	24.0	20.0
Reflector registers	✗	72.0	60.0
EMEM rings (same-island)	✓	152.0	126.667
EMEM rings (cross-island)	✓	168.0	140.0

work producer and consumer, with cross-island messages having the higher costs (table 5.1). This is comparable to message channels in the Rust and Go languages on commodity hardware (Tsilias, 2020).

To simplify implementation and to present a consistent API for other data-plane programs, packet headers are extracted and parsed using the tooling autogenerated by the P4 pipeline. This allows OPaL to handle configuration packets from the environment (whose protocol is covered later in appendix D) or elsewhere on-chip through the same mechanisms.

5.2.4 Intra-agent communication

Even with parallel problems such as *ParSa*, optimising for latency requires meticulous care in how work is passed out and aggregated. This is truer still when moving from the moderately fine-grained control of classical RL (~1 ms) to its logical limit (tens of µs). Ordinarily, the marshalling of requests, responses, and shared data access can incur significant overheads. On-chip execution and the nature of action preference computation allow us to use lockless atomic aggregation, removing the overheads of explicit messaging/packetisation. Moreover, adjacent functional units/cores often have special-purpose shared registers or share a small fast cache to accelerate communication. These capabilities underlie the design of the broadcast and aggregate primitives required by *Parsa*, and which are shown to some extent by fig. 5.4.

While I describe here how OPaL is optimised to enable the most efficient division of work, communication, and aggregation, these communication operations add per-task overheads in both the serial and parallel portions. Even in wait-free algorithms, this requires a minimum number of workers to improve upon the latency bounds of a serial approach. I investigate the exact worker count requirements imposed to break even or improve upon single-threaded execution in section 5.4.1.

Broadcasting OPaL's task broadcast implementation exploits the locality of cores in the NFP. Consider table 5.1: island-local IPC is considerably cheaper than the more generic methods we use for, e.g., the In and Out rings. The lowest-latency mechanism here, *next-neighbour registers*, allow for extremely quick communication between *adjacent* MEs, e.g., 0→1→2..., but their use limits us to a chain-forwarding approach. This remains, however, a net gain over arbitrary messaging via reflector registers in the absence of an actual broadcast bus. Consider the situation where 4 MEs are in use, thus the *controller* thread must notify 3 *minion++* contexts. In a chain forwarding scenario, MEs 1 and 2 receive commands sooner ($t = 20$ ns and 40 ns) than ME 3—and thus, all their contexts may start work sooner. Even assuming that all reflector writes can be sent in parallel, their use would enforce that all threads start at $t = 60$ ns. Reflector register IPC does remain a useful option for skipping ahead into chains of larger than 4 MEs, though these larger islands may only be used when an administrator is willing to replace one or more P4 pipelines. Inside of an ME, recall that all contexts share LMEM and a large register file. As such, work is passed out by copying the received message into a shared register region and simply notifying all other local contexts to awaken.

Aggregation Each context writes back to a single shared block of memory in CLS, performing atomic adds to a shared preference list and acknowledgement counter as required by the *Parsa* algorithm. The controller thread checks these whenever it is notified of task completion by a hardware signal fired after the acknowledgement counter is aggregated. This is essential for aggregation compared to the use of bounded message buffers, which caused significant head-of-line blocking in earlier implementations.

It is worth noting that CLS supports only 32 bit and 64 bit atomic arithmetic, and the native NFP register width is 32 bit. As a result, lower bit depth tile representations (8 bit and 16 bit) ultimately resolve to 32 bit atomic arithmetic. Is there a way to take advantage of this to gain additional throughput for these tile widths? That is, to find a bit-packing strategy which enables multiple additions to be performed in a single atomic operation as a makeshift form of SIMD? While this is doable, the main issue is that additions are performed on signed data in effectively an unsigned way, as the carry between arbitrary bit pairs cannot be disabled. *Unsigned* overflows are a common occurrence when operating on signed data in this manner[9]: naïve packing will cause the sign changes of adjacent computations to 'bleed into' neighbouring fields. In the non-atomic case a single packing bit suffices between any pair of values, all of which may be masked out in a single ALU operation. In the atomic case, n bit padding fields allow at most $2^n - 1$ additions from separate tasks without clearing. Having 136 tasks and 31 workers we require at least 8 bit padding to elide all atomic clears, or 5 bit if every addition is followed by a test-clear operation on all padding bits. Figure 5.5 shows the packing layouts in a 64 bit field which maximise the-

[9] Recall that `-1i8`→`0i8` is also `0xffu8`→`0x00u8`. As we can reasonably expect individual tile preferences for each action to oscillate between positive and negative, each sign change will trigger an unsigned overflow.

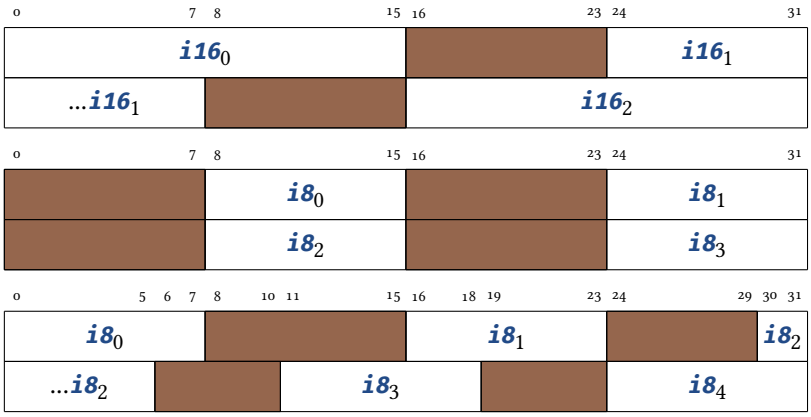


Figure 5.5: Bit-packing layouts within a big-endian 64 bit integer required to emulate SIMD addition using atomics. Packing requirements prevent the ideal 4 × and 8 × throughput gain for 16 bit and 8 bit data we'd expect from hardware SIMD. These layouts achieve 3 × , 4 × and 5 × ALU throughput, where the latter ***i8*** layout requires an explicit atomic clear after every addition. These maximise the count of byte-aligned data, though this cannot be guaranteed as we require at least $\log_2(workers)$ bits of internal padding.

oretical throughput—4–5 × for 8 bit and 3 × for 16 bit. Unfortunately, as related in section 5.3, this adds a consistent 10 % latency overhead to tile coded inference due to the additional non-atomic ALU operations needed to pack the input data. This can be useful on other platforms where atomic operations are much more expensive, or ALU use is cheaper.

Initialisation Any aggregation step requires us to first know the total number of workers. This can vary under the number of MEs running OPaL, the number of contexts assigned to each if CLS/CTM usage is required for another application, and because future designs may also introduce workers on other islands. This allocation of cores or chip area is set ahead of time by a framework or system administrator, but to enable greater runtime flexibility OPaL-*CoOp* agents enumerate themselves at runtime, during initialisation. To determine this, at startup the controller thread writes its own number of contexts into the first preference list entry, and passes on a message to its next neighbour. Each *minion++* then adds to this its own number of contexts, and forwards the message to the next ME in turn. The last *minion++* worker then increments the acknowledgement counter by 1. This count is then propagated back out to all contexts, acknowledged and awaited (for, e.g., local workset computation).

Parallels in other platforms On other SoC SmartNICs, we assume that similar MPMC communication channels and direct core-to-core messaging are possible under similar constraints, but note that this could be accelerated further by a true broadcast primitive. If the platform includes true message broadcast then implementation is simple. More specialised targets such as FPGA-based solutions may include an explicit bus between the controller

and all minion FUs to provide this in the most efficient manner possible. On NetFPGA, the writeback step can make use of native-width adders matching the tile data format, providing hardware SIMD acceleration as required.

5.2.5 Reconfigurability

OPaL allows policy design and learning parameters to be changed at runtime using at most two control packets. For instance, design changes are useful at the end of learning (moving from online to offline), or when trying to train a new policy for another problem from the same vantage point. Parameter changes are useful when an online agent must become more (or less) adaptive to new data (i.e., after detecting a changepoint in traffic). This extends to policy data, which may be imported from a pre-trained model via such packets and exported via PCIe to the host machine. Some aspects must be chosen at compile time; bit depth, parallelism strategy via *CoOp*/*Ind*, and maximum policy, tiling, or state sizes—these govern core operation or preallocated memory. Choosing a bit depth of 16 bit or 8 bit halves/quarters policy memory costs, allowing more complex problems to be modelled using more dimensions or fine-grained tiles.

In this implementation, configuration packets are carried over UDP and signalled to the P4 parser using a reserved pool 2 DSCP (Baker *et al.*, 1998) value, similarly to Y. Li, Liu *et al.* (2019). While this mainly automates parser generation, it also allows for configuration to be received from only trusted hosts (over the dataplane if needed) via P4 rules. The control packet generation library and evaluation frameworks which build upon it are written in Rust.

5.2.6 Work allocation

I use a simple first-fit work placement algorithm, algorithm 3, run in OPaL-*CoOp* whenever a full configuration is installed. This places the largest work item into the least loaded minion context of the least loaded ME, and assigns an equal number of tasks among all MEs where possible. Each work item is a separate *tiling* over a list of dimensions, where it can be reasonably assumed that these items are sorted by dimension count—thus, the reversed work list places the most computationally expensive tasks first. The approximate cost of any work item according to its dimension count and memory location was empirically measured offline and fed back into the scheduler—a mean 5.2 µs, 6.2 µs, 9.7 µs and 11.0 µs for bias (0D), CLS (1D), CTM (≤2D) and IMEM (≤4D) tilings respectively. Finally, this scheduler weighs the total cost per core based on the number of minion threads available. This weighting specifically accounts for the controller thread on the first core. To work around some fairly opaque waking behaviour between contexts on each ME, I apply a small penalty to minion contexts with a lower internal ID.

Algorithm 3: Task scheduling for ParSa

```
/* Assume we have a list ME_CTXS, which contains one entry
   for each included ME counting the number of its usable
   contexts, totalling N_CTXS. Also, assume we know the
   average cost of a task in each memory region, MEM_COSTS.
   */
1  struct Cost { cost, n_items, max_items };
2  Function Schedule work_items
3      let out = vec![vec![]; N_CTXS];
4      let alloc_sz = work_items.len() ÷ N_CTXS;
5      let alloc_spill = work_items.len() % N_CTXS;
6      let me_costs = [Cost { 0, 0, 0 }; ME_CTXS.len()];
7      let ctx_costs = [Cost { 0, 0, alloc_sz }; ME_CTXS.len()];
8      forall ctx in ctx_costs[..alloc_spill] do
9          ctx.max_items += 1;
10     forall ctx in ctx_costs do
11         let penalise_early_ctx = ME_CTXS[me_id(ctx)] -
             local_ctx_id(ctx) - 1;
12         me_costs[me_id(ctx)].max_items += ctx.max_items;
13         ctx.cost = penalise_early_ctx;
14         me_costs[me_id(ctx)].cost += ctx.cost;
       // Sort MEs by cost / ME_CTXS[me], breaking ties on
          largest id.
15     let me_heap = min_heap(me_costs);
       // Sort CTXs by cost, breaking ties on smallest id.
16     let ctx_heaps = [min_heap(ctx_costs[first_ctx(me)..ME_CTXS[me]])
        for me in 0..ME_CTXS.len()];
17     forall item in work_items.reverse() do
18         let me = find_min(me_heap);
19         let ctx = find_min(ctx_heaps[me]);
20         me.n_items += 1;
21         ctx.n_items += 1;
22         if me.n_items ≥ me.max_items then
23             me_heap.remove(me);
24         else
25             me.cost += MEM_COSTS[item.region];
26             me_heap.rebalance();
27         if ctx.n_items ≥ ctx.max_items then
28             ctx_heaps[me].remove(ctx);
29         else
30             ctx.cost += MEM_COSTS[item.region];
31             ctx_heaps[me].rebalance();
32         out[ctx].push(item);
33     return out
```

Naturally, for n tilings and m threads this procedure is $\mathcal{O}(n \log m)$: two find/update min operations into binary heaps per tiling, storing $m/8$ and ≤ 8 costs respectively. As an implementation detail, given that number of MEs is small we may replace their minheap with a list to reduce the setup overhead (i.e., constant terms) in exchange for worse asymptotic scaling. This also enables us to compare $cost_i \times ME\_CTXs[best] \leq cost_{best} \times ME\_CTXs[i]$ rather than repeatedly apply fixed-point division.

5.3 Evaluation

To evaluate OPaL fairly, we must investigate its performance characteristics. Primarily, these include raw latency and throughput statistics, co-existence with other dataplane programs on the same PDP device, validating my implementation and design choices, and assessing OPaL's impact on dataplane traffic. Here, I describe the experimental setup I use to evaluate these criteria. Moreover, I detail each of the individual experiments performed, including any defined baselines or more specific technical requirements.

5.3.1 Experimental setup

For traffic generation, and to employ a portfolio of CPUs at different performance points for comparison purposes, I use 3 different machine configurations to support the below experiments. These testing machines had the following hardware, all with 32 GiB RAM:

MidServer Intel Xeon Bronze 3204 (6 × 1.9 GHz),

HighServer Intel Xeon Silver 4208 (8 × 2.1 GHz),

Collector Intel Core i7-6700K (4 × 4.2 GHz).

Mid/HighServer are rackmounted server-grade hardware and are representative for situations where a server administrator aims to include inference in the packet path as part of a bump-in-the-wire deployment. Both of these servers ran Ubuntu 18.04.5 LTS (4.15.0-140-generic). *Collector* accounts for higher clocked consumer-grade hardware having fewer physical cores, capturing the case of mirrored (out-of-rack) traffic processing. This also enables the estimation of host performance when directed to a virtualised network function. *Collector* ran Ubuntu 18.04.4 LTS (4.15.0-96-generic)

OPaL and its firmwares were evaluated on server blade configurations (*MidServer* and *HighServer*), each with a single Netronome Agilio LX 1 × 40GbE SmartNIC (NFP-6480, 1.2 GHz). Firmwares were built to include a P4 toolchain using the default ME, context, and island assignment. Control programs were built using `rustc` version 1.52.1. OPaL's firmware is built to run

its RL logic on a 4-ME island of the NFP-6480, totalling 32 contexts. This is the largest cluster of cores which is not in use by a P4 pipeline. Versions of these firmwares using 32 bit, 16 bit and 8 bit arithmetic and registers to represent tiles, inputs, and values were built. Where feasible, I use these to test 32 bit, 16 bit and 8 bit quantised arithmetic.

All OPaL timing measurements were repeated over 10 000 state packets (preceded by 1000 warmup packets), retrieving item processing times over PCIe via the controller machine, from which throughput was derived. Host integer and floating-point performance numbers were acquired using a tile-coded Sarsa implementation written in numpy—throughput and latencies were measured over 10 trials of 10 s (with 5 s warmup/cooldown times). This differs from the NFP's methodology as a consequence of a numpy-based solution: Python's multithreading support is questionable at best due to the global interpreter lock, thus these agents must be run in parallel via separate processes. This *does* have some benefits for evaluation in that it also allows us to investigate the effects of oversubscription on host latencies.[10]

[10] Multithreading on NFP hardware prevents the costs associated with oversubscription even though contexts are not physical cores. Context switches are cooperative around I/O or voluntary wait points rather than time quanta, and the register file for each is preserved since the same program code is run by all contexts on an ME.

Policy sizes are set to those of the *Instant* DDoS control application introduced in chapter 4: 20-dim state vectors, a bias tile and 16 full tiling sets (7 × 1-dim, 8 × 2-dim, 1 × 4-dim), 8 tilings per set, 6 tiles per dimension, and 10 actions. As a reminder, such input state would contain various aspects of per-flow state (e.g., IATs, rates) which are combined with other state such as the last action taken (2-dim tilings) and loads along the ingress-egress path (4-dim). In *CoOp*, this creates 129 work items across 31 workers. Although the more successful *Guarded* agent design uses 3 actions, I choose a larger action set here to investigate the performance of more complex agents.

5.3.2 Experiments

Raw inference and learning performance I compare how long it takes for OPaL to compute actions and update its policy per state vector received, and report on the observed throughput of both firmware designs against floating point (numpy-based) implementations of Sarsa on a commodity host machine. This allows us to demonstrate the performance differences between the *Ind* and *CoOp* configurations, particularly in how *Ind*'s (and hosts') required policy locks impact throughput. We compare online learning performance (input states produce an output, and then update the policy) with offline (input states *only* produce an output) in these cases. Online performance marks the number of decisions that can be made per second (and associated latency) when training a policy. Offline performance is crucial for pushing a trained, known-good policy to agents in the network with an expected higher raw decision throughput. State-action latency is a shared property of both cases, with the main impact on throughput arising from the update step.

Building on this, I vary the amount of worker threads to show how OPaL

scales to fit available compute resources on a device. This is an important aspect for (automated) allocation of compute resources in an intelligent dataplane—particularly when cohabiting with other dataplane programs—and has effects on ahead-of-time work scheduling which are examined later. This also demonstrates the number of cores needed to achieve a given latency or throughput bound on a policy of representative complexity. Moreover, to demonstrate how these costs vary as policy complexity increases, I vary the number of total dimensions included in the tiling (i.e., the number of subtasks included in an inference or update step).

Work allocation I verify that the heuristic, runtime work scheduler described by algorithm 3—termed *Balanced*—makes meaningful use of the explicit memory hierarchy and cost of each work item. This is compared against several baselines, all of which allocate n_items_j tasks to every worker j as in algorithm 3:

- A *Naïve* chunked scheduler, which equally divides tasks among contexts. Each worker j visited in numerical order takes the first n_items_j free tasks.
- A *Random* allocation.
- A simple *Modular* allocator, designed to account for the fact that tasks with larger indices are typically more costly to execute, thus taking a relatively even spread of task indices. A worker j out of w takes $k = n\_items_j$ tasks, given by $work_j = \{j + i \times w \mid i \in [0, k)\}$. For instance, worker 4 of 31 (splitting 129 items) would take tasks 4, 35, 66, 97 and 128.

The *Naïve* and *Modular* schedulers have the benefit of being computationally simple—allowing each worker thread to independently compute its own allocation without issue. This has some impact on dynamic reconfiguration of an OPaL agent, namely on the amount of serial and distributed work required in response to a policy structure change. The *Random* allocator is useful for evaluation in that it considers many separate (though unlikely) schedules, allowing us to determine whether there exists any available improvement.[11] When measuring schedule effectiveness, ParSa is timed from the start of Ctl until an action is produced (*Action*, lines 12–17), the serial portion of update state management is completed without triggering an update (*Update Prep*, lines 12–26 given $\neg(found\_s \wedge found\_r)$), and the procedure finishes (*Update*, lines 12–26 given $found\_s \wedge found\_r$). In all measures I use maximum-size 32 bit policies as described earlier.

[11] Obviously this cannot explore the *entire* schedule space of $N!$ permutations, but this does give us sensible measures for the *expected* schedule cost to meet and can observe better lower bounds.

End-to-end RL latency I compare the key RL decision-making latencies we discuss in fig. 3.7 across 3 scenarios: completely in-NIC (OPaL), delegating RL decisions to a SmartNIC's controller machine, and using a VNF on the

same machine for RL inference. To enable this, I combine the raw latency metrics of this system with accurate PCIe and VNF framework costs offered by existing work.

Co-existence with the dataplane While varying the rate of full RL updates performed by OPaL-*CoOp* (32 bit) from 0–16 000 actions/s, I measure packet losses and sample latencies of cross traffic forwarded over a P4 pipeline hosted on our SmartNICs. This allows us to quantify whether on-chip (out-of-path) execution impacts ordinary dataplane behaviour through indirect means: e.g., EMEM cache evictions or hidden resource contention.

I perform these tests using Pktgen-DPDK (Wiles, 2021), placing an NFP in *MidServer* as the device under test and connecting *HighServer* over a 40 Gbit/s direct copper cable as the traffic source via the default NFP firmware. Throughput and loss tests are performed using 7/1 transmit/receive queues at 100 % send rate for 10 bursts of 30 s, and perform latency tests using 1/1 transmit/receive queue at 10 % send rate for 200 000 measurements (sampling at 2000 Hz for 10 × 10 s). This provides maximum throughput in the former case (relying solely on NIC counters for loss counting). In the latter case, this minimises host resource contention to observe exact latency measurements, have a high enough sample count to detect subtle (aggregate) latency effects, and eliminate *host* receive drops. DPDK was setup using 4 × 1 GiB hugepages. Sent traffic was comprised of fixed-size 64–1518 B packets (Bradner & McQuaid, 1999). CPU clock scaling was disabled on *HighServer* to enable more accurate latency measurement.

Resource requirements Using the maximum policy size defined above, I investigate how the memory requirements imposed by OPaL vary with the number of dedicated MEs, over and above a base P4 forwarding plane. I report resource use for 32 bit *Ind* and *CoOp* agents, with hash-tables sized to 4096 state-action pairs and 16 separate reward values. Firmwares are compiled to make use of 1 and 4 MEs. This captures the relative cardinality of network RL traces to rewards, as many input flows will typically map to one or few reward values (i.e., DDoS attack size estimation per egress-AS, queue occupancy in the case of AQM per output port).

Deployability By timing agent setup and compile times, I measure the runtime costs needed for an administrator to repurpose an installed agent in a live network. These include the costs of changing hyperparameters or policy structure data, mainly incurred by regenerating caches of parameters used in tile hit computation. In the case of *CoOp*, this includes measurements of the cost of policy schedule computation as a function of workers and tasks. Beyond this, I relate the costs of more complex reconfiguration in an NFP-based system, including firmware installation times required to

swap between *CoOp* and *Ind* models, in addition to compile costs which some agent changes may mandate.

Magnitude comparisons against PDP ML While the work in this chapter presents the first in-NIC RL methodology, I offer a rough comparison against existing in-NIC ML inference—principally BNN-based and MAT works. This comparison mainly considers inference latency as a function of input data size between OPaL and each competing approach, keeping in mind the target environment of each (ASIC, FPGA, or NFP SmartNIC).

5.4 Results and discussion

Investigating the performance of OPaL compared to classical RL techniques executed on commodity host machines, we see that *CoOp* offers a 15–21 × speedup in median–99.99$^{\text{th}}$ state-action latency as well as 9.9 × greater online learning throughput. Crucially, in-NIC execution offers tight tail latency bounds compared to host-based approaches. I report on how OPaL scales as additional device resources are added, noting that both in-NIC designs outperform commodity hosts using just one core in latency and online throughput. Furthermore, *Ind* provides higher per-core offline throughput than host-based approaches, even though our measured hosts exhibit higher clock speeds. Finally, I show that OPaL has minimal impact on data-plane cross-traffic carried by its parent device, and that it performs at a similar level of performance as other PDP ML which are implemented as MicroC programs.

5.4.1 Raw inference and learning performance

Table 5.2 shows how OPaL compares in latency with a numpy-based RL policy. *CoOp* achieves sub-35 µs median latency, with 99$^{\text{th}}$ and 99.99$^{\text{th}}$ percentile latencies less than 1 µs worse using 4 MEs of the NFP-6480. This corresponds to 15–21 × speedups over a *Collector* host using floating-point arithmetic. Importantly, *Ind* still achieves lower median state-action latencies (2.79 ×) *and* update times (2.63 ×) than a dedicated *Collector* host while requiring only a single core or dedicated functional unit. Note that the numpy-based integer results underperform compared to the floating-point variant—median action latencies are 14.6 % worse, with 7.9 % longer update times.

Table 5.3 compares OPaL's throughput against host-based execution. The worker count was chosen for host machines such that their throughput was maximised without causing latency degradation. This equalled the amount of physical cores on each device—moving beyond this (even below the number of hyper-threads) would hamper tail latencies by an order of magnitude.

Table 5.2: Latencies and computation times for OPaL versus commodity hardware hosts. On-device execution is crucial in not only lowering latencies, but in reducing tail latencies. Lower is better, with the best marked *in bold*.

Datatype	Machine/FW	State-Action Latency (µs)			State-Update Time (µs)		
		Median	99^{th}	99.99^{th}	Median	99^{th}	99.99^{th}
Float	Collector	515.94	606.06	725.03	606.06	636.82	833.99
	MidServer	1069.07	1125.1	1508.0	1260.04	1605.99	1719.864
NpInt32	Collector	562.91	668.05	889.06	653.03	715.02	943.9
	MidServer	1154.9	1202.11	1595.252	1362.09	1477.0	1836.657
NpInt16	Collector	562.91	647.07	831.13	653.98	782.01	952.01
	MidServer	1152.99	1234.05	1607.932	1361.13	1415.97	1811.071
NpInt8	Collector	564.1	645.88	861.008	651.84	739.1	922.012
	MidServer	1152.04	1204.97	1602.022	1361.13	1502.99	1818.934
Int32	OPaL-*Ind*	185.133	185.533	186.213	230.840	231.347	232.227
	OPaL-*CoOp*	**34.347**	**34.520**	**34.573**	**62.000**	**62.440**	**63.120**
Int16	OPaL-*Ind*	193.427	193.787	194.587	240.333	240.840	241.560
	OPaL-*CoOp*	36.147	36.240	36.280	64.667	65.080	65.973
Int8	OPaL-*Ind*	194.520	194.840	195.240	241.173	241.707	242.760
	OPaL-*CoOp*	36.227	36.307	36.347	64.333	64.867	65.693

Table 5.3: Action and update throughputs for OPaL versus commodity hardware hosts. Most designs cannot scale online performance with additional cores. Higher is better, with the best marked *in bold*.

Datatype	Machine/FW	Workers	Throughput (k actions/s)		Throughput/core (k actions/s)	
			Offline	Online	Offline	Online
Float	Collector	4	7.673(49)	1.627(31)	1.918(12)	—
	MidServer	6	5.584(30)	0.791(12)	0.931(5)	—
NpInt32	Collector	4	6.960(131)	1.493(57)	1.740(33)	—
	MidServer	6	5.176(15)	0.727(8)	0.863(3)	—
NpInt16	Collector	4	6.973(116)	1.495(37)	1.743(29)	—
	MidServer	6	5.182(35)	0.736(10)	0.864(6)	—
NpInt8	Collector	4	6.968(101)	1.520(26)	1.742(25)	—
	MidServer	6	5.190(28)	0.731(16)	0.865(5)	—
Int32	OPaL-*Ind*	32	**172.875(229)**	4.333(5)	**5.402(7)**	—
	OPaL-*CoOp*	32	29.166(173)	**16.141(73)**	0.911(5)	**0.504(2)**
Int16	OPaL-*Ind*	32	165.437(118)	4.161(4)	5.170(4)	—
	OPaL-*CoOp*	32	27.664(36)	15.471(54)	0.865(1)	0.483(2)
Int8	OPaL-*Ind*	32	164.524(142)	4.147(5)	5.141(4)	—
	OPaL-*CoOp*	32	27.631(101)	15.552(68)	0.863(3)	0.486(2)

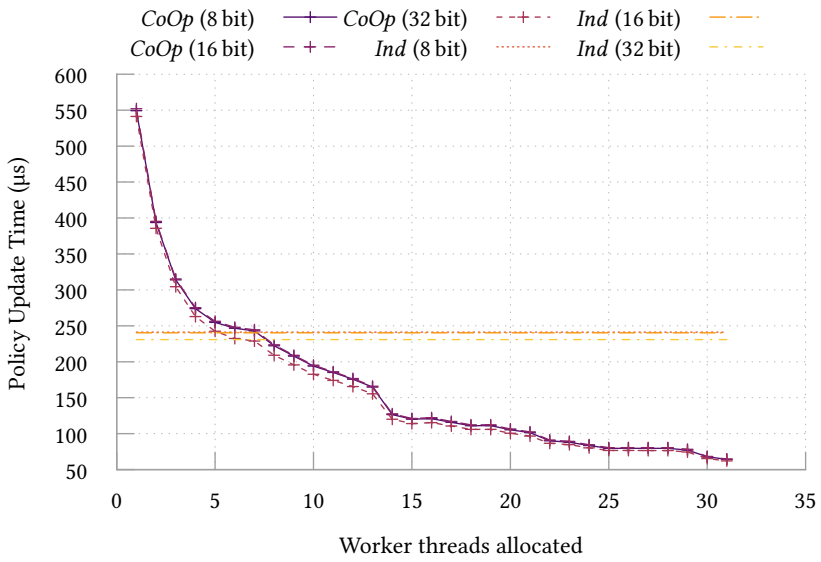


Figure 5.6: OPaL's combined update and inference time as the degree of parallelism is varied. *CoOp*'s online learning performance improves with additional cores, on max size tasks (129 work items). This requires 8 workers to offer greater online throughput than single-threaded in-NIC RL. Sharper performance increases occur when a new physical core is added (7–8) or the scheduler works around a bottleneck (13–14).

To make the comparison fair in the context of many-core CPU environments, I include per-core throughput. *Ind* achieves 2.82 × higher offline throughput than commodity *Collector* hardware in spite of the NFP-6480 having a considerably slower clock speed (0.29 ×). When compounded with the abundance of such weaker chips, in-NIC RL is able to deliver much higher throughput. As anticipated, the *CoOp* strategy is key in achieving serviceable throughput in an online learning agent, 9.9 × that of a dedicated collector machine, as the write lock around policy updates creates a bottleneck.

By limiting the available workers in software, I show how *CoOp*'s policy update time (thus online throughput—fig. 5.6) and state-action latency (fig. 5.7) scale with available cores. While *CoOp* always outperforms the host-based floating point implementations according to the earlier findings, we observe that there are two distinct crossover points which must be met to overcome our own *Ind*; 8 workers for online throughput, and 3 workers for state-action latency. Some artefacts of the hardware environment and design choices are visible, such as the addition of new physical cores being more significant than contexts, and the presence of some schedule bottlenecks. Most importantly however, *CoOp*'s resource demand is tunable at compile time to meet the online training rate and action latency required by a task or environment.

Figures 5.8 and 5.9 show how policy complexity affects update cost and state-action latency respectively, scaling from a bias tile up to the full DDoS policy

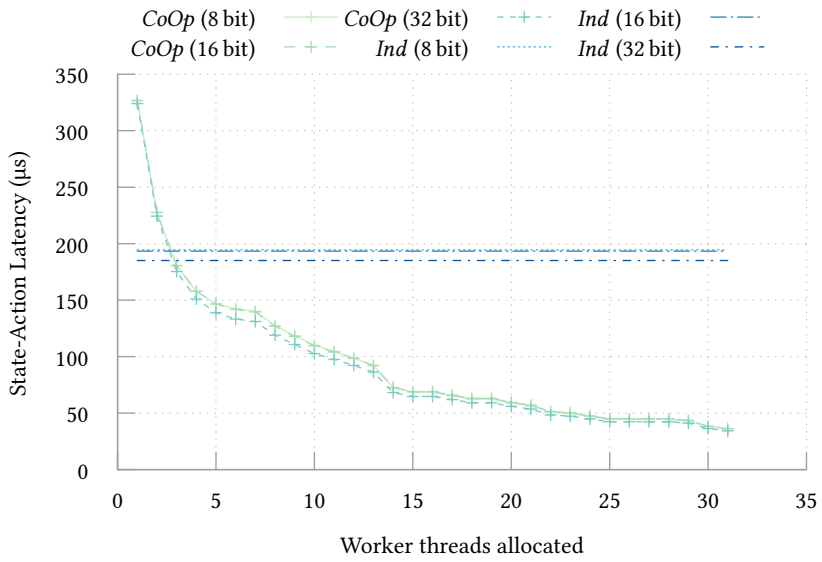


Figure 5.7: OPaL's state-action latency as the degree of parallelism is varied. *CoOp*'s action latencies similarly improve with more cores. This requires 3 workers (4 total contexts) to improve upon the state-action latency of single-threaded in-NIC RL.

size.[12] *CoOp* always produces an action in less time than *Ind*, but requires at least one state-based tile to excel in online learning. Note that this is a trivial case, as using *only* a bias tile returns a single preference list regardless of input state.

[12] Input vectors here all have 20 elements regardless of the policy design.

A key aspect of in-NIC execution is that it allows far tighter bounds on tail latency compared to host inference. Examining the state-action latencies in table 5.2, we see that 99.99$^{\text{th}}$ percentile latencies exceed the median by 0.58 % and 0.66 % for *Ind* and *CoOp*, respectively. Similar results were observed for other bit depths. By contrast, host-based tail latencies are at least 40.53 % greater even when the parallel worker count is at or below the physical core count. The cumulative distributions of these are shown in detail by fig. 5.10, noting how just one additional CPU-intensive task—potentially automated system updates, scans, or another VNF/traffic processing task—impacts tail latencies further (*Float(Over)*).

A noteworthy trend is that 8 bit and 16 bit versions of OPaL consistently underperform compared to 32 bit, except for smaller workloads (seen in the zoomed portions of figs. 5.8 and 5.9). This occurs even though our implementation is optimised to read and write policy data in batches (achieving fewer I/O operations). We see this because the native register width on the NFP is 32 bit, and so the compiler must emit extra instructions around arithmetic operations to correctly load and update values. This matches 32 bit becoming dominant in complex workloads: higher dimension tilings require more arithmetic operations. Most of the I/O comes *after* this step, causing ALU use to dominate. This also explains why 32 bit becomes the best choice at different policy complexities for online (fig. 5.8, 10 dims) and

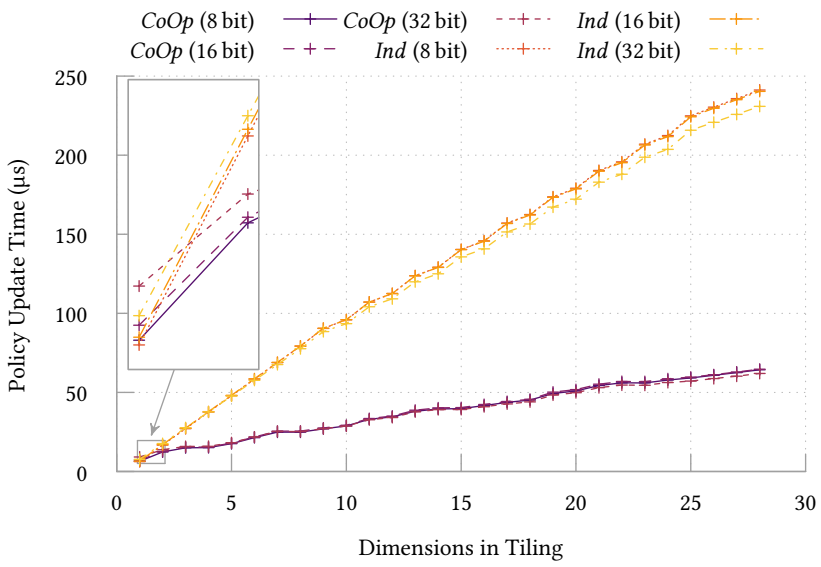


Figure 5.8: OPaL's combined update and inference time as the number of tiling dimensions is varied. *CoOp* fully processes updates faster than *Ind*—thus has higher online performance—on almost all policy sizes. Lower bit depths are more effective on simpler policies.

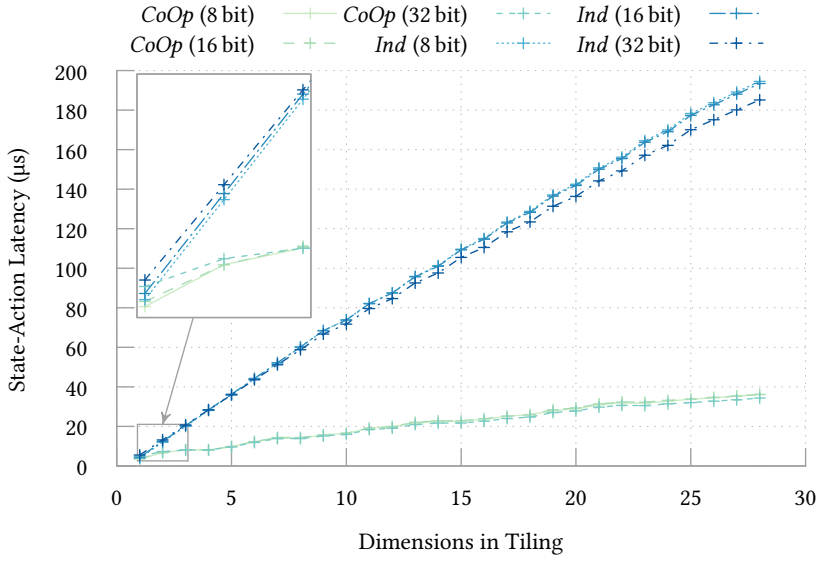
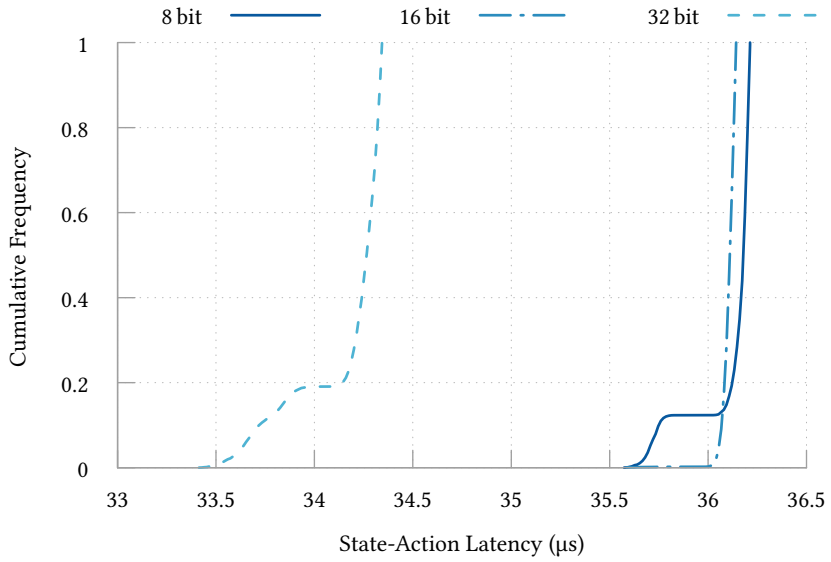
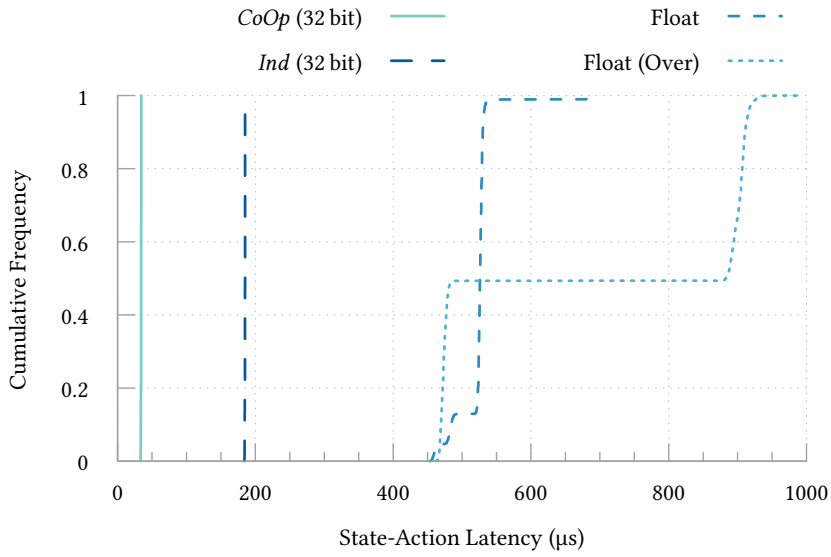


Figure 5.9: OPaL's state-action latency as the number of tiling dimensions is varied. State-action latency scales with additional work in a similar manner to overall processing time; though 32 bit firmwares become more effective sooner (at 3 input dimensions, or 17 tasks).



(a) *OPaL's* CoOp *design achieves consistent, tight latency bounds, with less than 1 µs between minima and maxima for all choices of bit depth. In most cases, latencies follow 32 < 16 < 8 bit.*



(b) Ind *observes higher—though similarly tightly bounded—latencies than* CoOp. *In hosts tail latencies suffer, particularly when oversubscribed as shown by Float(Over) which adds a single additional worker above the host's physical CPU count.*

Figure 5.10: Cumulative state-action latency plots for OPaL and host-based execution. In-NIC execution offers lower (and more consistent) latencies than execution on a numpy-based host. 32 bit inference is the lowest-latency strategy in OPaL.

offline (fig. 5.9, 3 dims) agents, where hashtable accesses and a *memcpy* of the state vector fall into the serial portion of the online algorithm. Smaller bit depths still give a 2–4 × saving in memory for policy storage (enabling more fine-grained or complex policies), in exchange for slightly worse latency and throughput. To overcome this, I investigated SIMD-like bit-stuffing of values into a single word during atomic writeback (as the platform offers both 32 bit and 64 bit atomic addition) as presented in section 5.2.4. Unfortunately, manipulating tiles into the correct format added 10 % extra overhead.

5.4.2 Work allocation

Figure 5.11 shows that our heuristic allocator (*Balanced*, algorithm 3) is key in achieving consistent sub-35 µs and 65 µs latencies and update times, respectively. The trend is repeated for all bit depths. The constant difference between *Action* and *Update Prep* scales with bit depth, matching storage and lookup work in the serial portion of *ParSa* (fig. 5.12). The severe underperformance of the *Naïve* allocator confirms that work item complexity is correlated with its index, as batching work in contiguous chunks gives some workers only high-dimensionality tilings. The minor gap in lower bound performance between the *Random* and *Balanced* allocators suggests that further optimisations can be made. I expect that closing or exceeding this gap may require more complex modelling of hardware thread interactions, which lies far beyond the scope of in-NIC scheduling. Some additional complexity may be tolerated, subject to code store limits—scheduling runs exactly once per configuration change, so does not impact per-action code.

An interesting aspect of *CoOp* and *ParSa* is that adding cores has both diminishing returns and key thresholds to pass. Consider fig. 5.13, where the throughput per worker decreases with cores but occasionally increases sharply. Later downward ticks (25–29 workers) correspond to plateaus in throughput. This is a problem stemming from the granularity of work items (i.e., tilings in *ParSa*), where our scheduler cannot find a better solution to a bottleneck until extra cores are allocated. As mentioned in section 5.2.6, individual state-action computation work items were measured as taking a mean 5.2 µs, 6.2 µs, 9.7 µs and 11.0 µs for bias, CLS, CTM and IMEM tilings respectively. Though we have a 4.2 × factor of task oversubscription it is clear that as the count of worker threads increases, latencies are eventually bound below by the length of the longest task.

5.4.3 End-to-end RL latency

Referring to fig. 3.7, we take t_2 from table 5.2 for host and in-NIC processing times, and substitute $t_1 + t_3$ for the state packet round-trip time to the decision site:

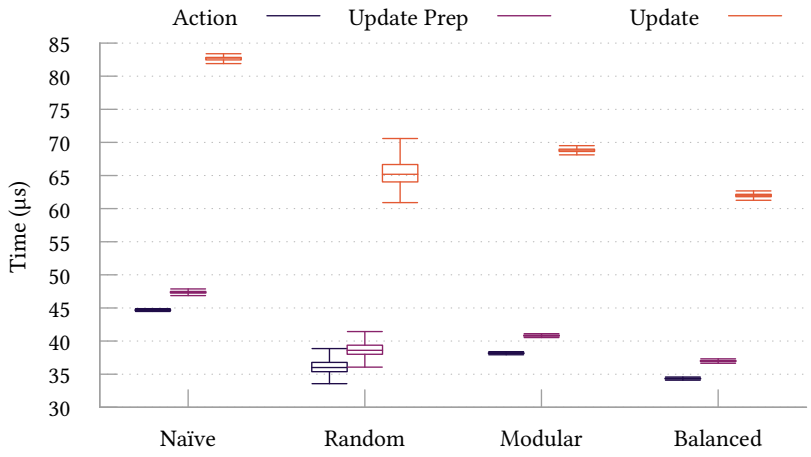


Figure 5.11: Action and update compute times in a 32 bit *CoOp* agent under different work schedulers. The *Naïve* and *Modular* schedulers achieve worse performance than *Random*'s median value. The *Balanced* scheduler introduced in algorithm 3 outperforms these and achieves tight latency bounds while outperforming the expected random schedule. However, there is still a slight performance gap between the empirically observed minimum cost and *Balanced*.

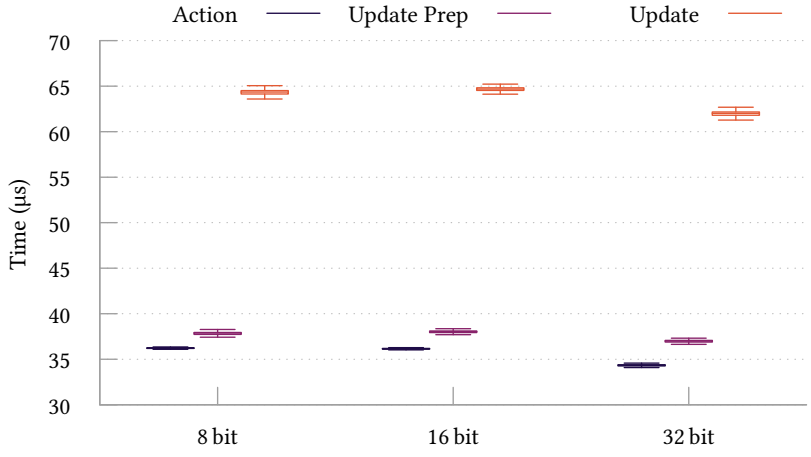


Figure 5.12: Stage times of OPaL's *Balanced* allocator at 8 bit, 16 bit and 32 bit depths. While the core inference and update logic becomes more efficient at higher bit depths, the logic in the post-inference serial portion (i.e., *Action→Update Prep*) becomes marginally less taxing at smaller bit depths. These serial portions consume 1.653 µs, 1.907 µs and 2.653 µs for 8 bit, 16 bit and 32 bit respectively.

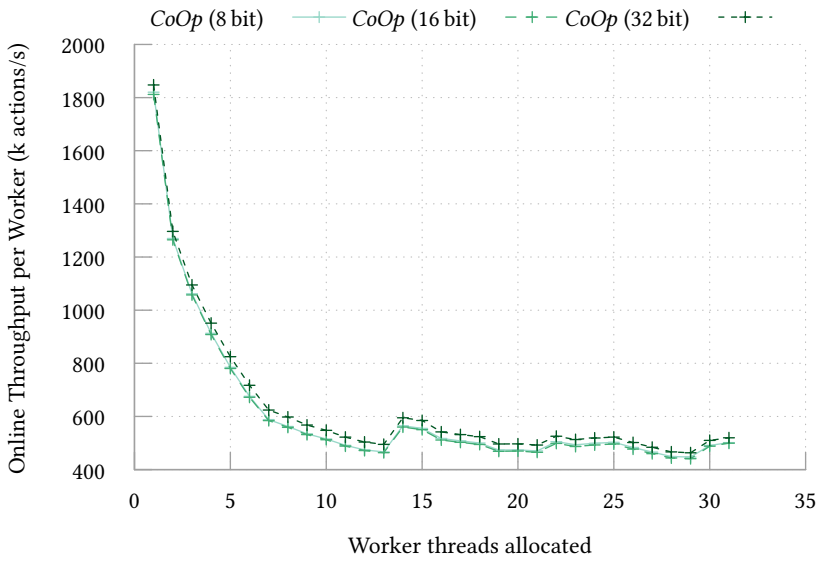


Figure 5.13: OPaL-*CoOp* online throughput per core as the number of worker threads is varied. Additional threads exhibit diminishing returns on overall throughput for a given policy size. Note that the spikes of increasing per-core value correspond to the scheduler bottlenecks observed in fig. 5.6.

In-NIC. As described in section 5.2.3, EMEM rings have a median one way delay cross-island of 140 ns, giving a *median 34.63 µs end-to-end inference latency.*

Dedicated Collector. Offloads hosted in this manner will employ DPDK to maximise performance, giving one-way PCIe delays of 0.9–2.3 µs for network packets (Neugebauer *et al.*, 2018). A UDP packet carrying 20 elements of state in OPaL is 128 B, so costs 1 µs to forward, and the reply state-action pair is slightly larger. *This gives an end-to-end inference latency of 517.9 µs.*

VNF Offload. Cziva and Pezaros (2017) show that lightweight VNF frameworks like GNF (Cziva & Pezaros, 2017) and ClickOS (Martins *et al.*, 2014) add 45–55 µs *additional* RTT latency above PCIe costs, which remain an estimated 1 µs as above. *This gives an end-to-end inference latency of 572.9 µs.*

Using these estimates, in-NIC RL inference offers 14.96 × and 16.54 × speedups in latency over collector and VNF deployments respectively (assuming no steering cost in either case). The RTTs in our earlier microbenchmark—fig. 2.5—are slightly more complex to compare against; they include the MAC times, wire times, and send-side costs of an additional host machine (a minor form of steering). What is clear, however, is that raw OPaL inference costs are very similar to those required to reach a host machine via XDP and its related technologies[13] which have sprung up in more recent years. We also contrast these against DRL policies on network tasks, which can

[13] To reiterate, the median RTTs were around 19 µs for DPDK, 30.25 µs for Native XDP, and 32.5 µs for `AF_XDP`.

take 3 ms to compute (Sivakumar *et al.*, 2019)—2 orders of magnitude above OPaL with identically sized (20-dim) input vectors.

5.4.4 Co-existence with the dataplane

The traffic generation setup met 40 Gbit/s for packet sizes ≥256 B. For frame sizes of 64 B and 128 B, input traffic rates were 17.4 Gbit/s and 32.9 Gbit/s respectively (33.9 Mpps and 32.2 Mpps). Passing this traffic over the NFP device running OPaL, no packet losses were incurred at any rate of RL actions.

Figure 5.14 shows the effect of RL workloads on the round-trip latencies of cross traffic. As observed latencies do not obey a normal distribution (particularly 256 B and 1518 B, which are bimodal), I employ a one-tailed *Mann-Whitney U test* to mark statistically significant population increases in latency ($p < 0.05$) with a "+". In general, statistically significant latency increases concentrate around smaller packet sizes. All (bar one) of these affected 99$^{\text{th}}$ percentile latencies by under 0.38 % (at most 78 ns). This slight degradation can be explained by increased pressure on the NFP's *Command Push-Pull* (CPP) bus, which is responsible for handling cross-island accesses to memory (particularly IMEM/EMEM) and other resources. OPaL places load on the CPP bus through its In/Out EMEM rings and last-tier policy accesses. This also explains the sensitivity of 256 B packets to OPaL—the NFP P4 toolchain segments packets, storing metadata (e.g., MAC prepend) and the first bytes of a packet in a 256 B CTM block while parking their payloads in EMEM. 256 B packets overshoot this due to metadata, causing small I/O accesses at a high rate for packets sized around this cutoff.

The anomalous result is 128 B packets at 3000 RL action/update computations per second, causing a 222 ns (1.18 %) increase, shown in fig. 5.15. This is observed through a shift of some packet latencies from the mode towards the tail, but no other changes in the distribution. In the above context, we believe that the inbound request rate is weakly synchronised with inbound packets, causing a higher level of burstiness around accesses to the CPP bus. I expect that dedicated hardware or FPGA designs can avoid this problem by having dedicated In/Out access mechanisms for an OPaL agent, which would of course completely eliminate the possibility of resource contention.

5.4.5 Resource requirements

Table 5.4 shows how OPaL consumes shared memory as it scales to fit a device's compute resources, compared with a simple P4 forwarding application. As one program is installed per ME, these results represent the minimum and maximum resource use on a single island (i.e., without replacing P4 workers). We observe negligible costs on shared EMEM (~4 MiB),

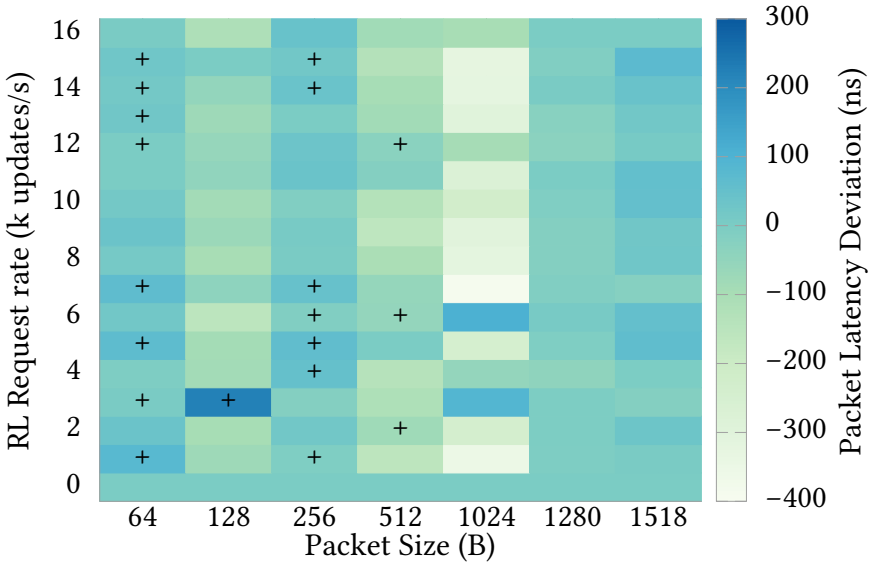


Figure 5.14: Deviations in 99th percentile cross-traffic RTTs for an OPaL agent processing 0–16k updates/s. Statistically significant increases in population latency are marked with a "+". These concentrate on smaller packet sizes, and are typically sub-78 ns.

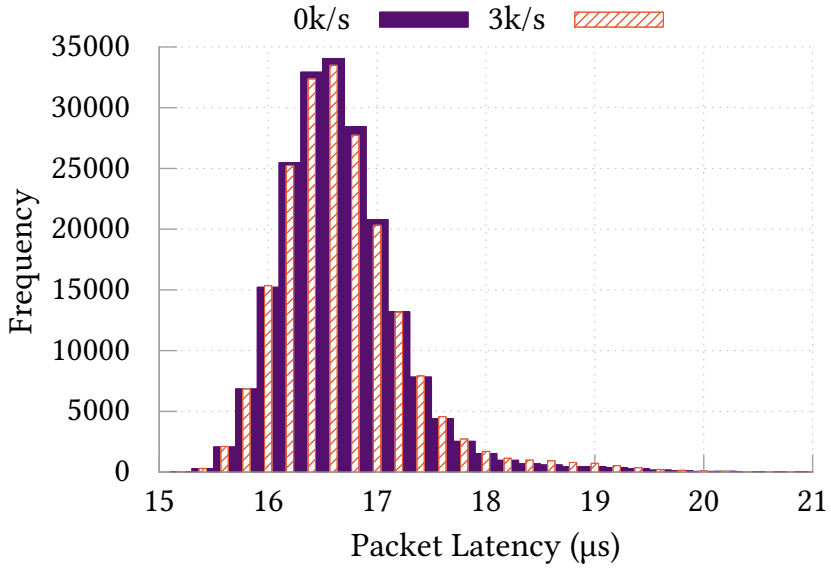


Figure 5.15: Distribution of RTTs for 128 B packets for 0 and 3000 full RL updates/s. This is an anomalously high deviation present in fig. 5.14—the modal value has not altered, but the peak has dampened somewhat, increasing tail latencies.

Table 5.4: NFP memory cost incurred by OPaL when built to use 1 and 4 MEs (32 bit). CLS and CTM are shared between all programs on the same island (placing our RL agent on i5), while EMEM and IMEM are shared between all NFP programs on a NIC.

Firmware	EMEM		EMEM Cache		IMEM		i5.CLS		i5.CTM	
	MiB	%	KiB	%	KiB	%	KiB	%	KiB	%
Base P4	6776.67	88.24	268.52	2.91	858.28	10.48	0.00	0.00	0.00	0.00
Ind(1)	6780.21	88.28	2541.08	27.57	1263.28	15.42	24.75	38.67	94.25	36.82
Ind(4)	6780.22	88.28	2545.33	27.62	1263.28	15.42	51.18	79.97	107.00	41.80
CoOp(1)	6779.12	88.27	1773.59	19.24	1263.28	15.42	22.41	35.01	90.00	35.16
CoOp(4)	6779.12	88.27	1769.84	19.20	1263.28	15.42	52.16	81.49	90.00	35.16

incurred due to hashtables for past state and rewards. The most significant costs arise due to policy data (405 KiB shared IMEM, 90 KiB local CTM, 15 KiB local CLS), which can be halved or quartered using 16 bit and 8 bit quantisation and remain constant regardless of compute unit usage. This is a high upfront cost on per-island resources (CLS/CTM)—OPaL leaves resources for other off-path dataplane applications, but is fairest from 3 cores onwards.

5.4.6 Deployability

Setup of OPaL uses two packet types: *setup*, which contains learning parameters, hyperparameters, and most aspects of a policy, and *tiling*, which provides a list of state indices for tiling sets. Instrumentation in OPaL found that setup and tiling packets take a mean 27.03 µs and 16.69 µs to be processed on *Ind*, allowing an agent to be swapped between online and offline operation (or repurposed for another task) painlessly. Online/offline swaps are useful when an agent should cease learning (i.e., when performance has converged), or if a change in the environment suggests that more training is needed. Tiling packet processing scales linearly with the number of *tiling sets*, due to the needed precaching of tile set boundaries. Online-offline swaps for *CoOp* exhibit similar cost, however the need for explicit scheduling means that policy/tiling *structure* changes (including the first complete setup) take 422.63 µs for the full-size policy described above. The time taken for *CoOp* to schedule its tasks was found to scale with the number of workers (m) and work items (n) as described earlier (section 5.2.6, $\mathcal{O}(n \log m)$). Ignoring the trivial solutions, reducing the worker count to 1 costs a mean 238.22 µs, while placing a single task incurs 53.79 µs. Policy *data* changes require no additional work in any case, resolving purely to `memcpy`s.

Firmware (re-)installation (i.e., changing from *Ind* to *CoOp*, bit depth, or increasing maximum policy sizes) took a mean time of 38.83 s. In the event that appropriate firmwares are not pre-compiled, compiling and linking both OPaL and the P4 toolchain took around 35 s, while changing only OPaL parameters required around 25 s. These results show that OPaL can be easily adapted and altered by network administrators once in place, and illustrates

an advantage of SoC-based SmartNICs.

5.4.7 Magnitude comparisons against PDP ML

Full, in-network packet tagging and classification by pre-trained BNNs is shown by *N3IC* (Siracusano *et al.*, 2020). N3IC achieves packet inference in 45 µs on the NFP, and 0.3 µs on NetFPGA for 256 bit inputs. Comparatively, OPaL-*CoOp* can process an identically-sized input (8-dim) in a median 13.83 µs. This work handles larger inputs (640 bit) at lower latencies (34 µs), and offers online learning. I expect that a NetFPGA implementation of OPaL would enjoy a similar factor of speedup. No works investigate the runtime training of BNN function approximators in-NIC.

Langlet (2019) has shown the viability of NN inference using 64 bit quantisation on the NFP, using *in-path* compute rather than our asynchronous model. Inference latency on small networks (3 layers, 30 neurons) can be as high as 500 µs on line rate traffic using 2304 bit inputs, emphasising the value of path-adjacent compute.

Taurus (Swamy *et al.*, 2020) proposes that efficient line-rate inference can be achieved using a configurable grid of map-reduce units in the packet pipeline (implementing e.g., LSTMs and SVMs). On CGRA hardware, they achieve sub-µs extra latency for inputs of an unclear size. *IIsy* (Xiong & Zilberman, 2019) shows how *classical ML inference* (SVMs, Naïve Bayes, etc.) can be converted into match-action tables compatible with *any* P4 deployment. They achieve mean 2.62 µs extra in-path latency on NetFPGA (at line rate in most cases) on 59 bit inputs when illegal values are excluded. As these approaches are reliant on specialised hardware-accelerated primitives for inference, an apples-to-apples comparison is difficult. However by considering the relative performance of N3IC between Netronome and FPGA implementations it is reasonable to place N3IC (and thus OPaL) in the same performance band.

5.5 Potential integrations

To show the applicability and use of OPaL, we propose an ideal integration which would benefit from in-NIC RL; online DDoS attack mitigation. I support this using other state-of-the-art P4/PDP developments, and discuss how best to balance the concerns of online training (*CoOp* agents) with throughput (*Ind* agents) in widespread deployment.

5.5.1 In-network DDoS defence

Classical RL can enable real-time, adaptive DDoS mitigation, as I discuss through chapter 4. The *Guarded* agent design uses a mixture of global

network state and local, per-flow state to monitor how flows respond to bandwidth changes and packet drop—applying the observations made by SPIFFY (Kang *et al.*, 2016b) (observing how flow behaviour reacts to a change in rate limits) with more allowance for congestion-unaware protocols. Actions then move flows up or down in punishment levels according to a finite state machine.

Why in-NIC? This approach relies on co-hosting traffic measurement and analysis alongside OpenFlow-compatible switches at the edge nodes of an AS. However, packet mirroring and moving RL computation to a host (potentially over layers of virtualisation) are all sources of additional, consistent state-action latency. Both traffic statistics collection *and* data-driven learning must be executed on such hosts and NFs. Unless running these stages in a dedicated pipeline (adding further processing latency), resource contention between these processes will further impact tail latencies. Naturally, this requires high-performance hardware to be stacked at network egress points, potentially beyond reasonable space, power, or ventilation constraints. The solution to implement and improve upon this work using OPaL is to place its RL agents on SmartNICs at AS edge nodes—a bump-in-the-wire deployment.

Inputs To collate the required inputs and state, let us examine the recent innovations of the community. Low-latency, pure-P4 solutions to extract and record per-flow TCP state directly in the dataplane such as Dapper (Ghasemi *et al.*, 2017) and Sonata (Gupta *et al.*, 2018) are well-studied. In fact, the statistics offered by Dapper are a super-set of the local input state values employed by *Guarded* agents, offering an opportunity to further improve their efficacy. I propose placing such monitors in the P4 dataplane, existing on-chip alongside the OPaL agent. The required global state (load measures from network paths) must still come from elsewhere in the network; this is now the element at highest risk of becoming stale, but the least likely to vary significantly in response to individual actions.

The *Guarded* design as I present it uses theoretical "ground truth" rewards, whose correct implementation and designs were left as an open challenge. I posit that INDDoS (Ding *et al.*, 2021), which uses Count-min Sketches to estimate DDoS victim cardinality, could be an effective reward function source—i.e., using the number of detected victims as a loss value.

Integrating OPaL Before each monitoring action, we require that the table hosting this hybrid solution polls OPaL's Out ring for any generated actions. As noted in section 5.1.2, these actions would be placed into a small hash table and simultaneously exported to the attached controller to be inserted as P4 rules in batches. Afterwards, packet ingress timestamps would be used to emulate the TRS scheduler used by the anti-DDoS agents for

rate-controlled work, where state vectors would be selected and passed into OPaL. By design, active flows which are not *judged* after a configurable time are discarded to prune the work set and allow new flows to be seen by the agent. The tight bounds on execution time known *a priori* make it easy to calculate the maximum number of decisions which can be made per deadline. Reward values would then be separately inserted by a modified IND-DoS table.

Reducing state-action latency (i.e., with *CoOp*) is useful for minimising the noise inherent in learning. However, an agent is limited by the fact that it can take at least one RTT for meaningful changes to occur in a flow's behaviour ($\mathcal{O}$(ms) in a transit AS or ISP). Accordingly, this use-case benefits most from an increase in *throughput* using *Ind*. In this context, higher throughput means that network flows are more likely to be judged in every timestep, even when flow cardinality is high—making it more likely that changes in flow behaviour will be observed, acted on, and learnt from.

Note that the TRS scheduler is designed to handle large numbers of attack flows, combining seen state vectors over time while the asynchronous agent is itself busy. By design, a number of (attack) flows beyond the maximum throughput simply makes it take longer in expectation for a flow to be re-assessed. As shown in section 5.3, OPaL far exceeds the throughput of host offloading. The control plane can then use wildcards or specific matches to narrow down or expand the set of flows to be controlled dynamically, though explicit TRS scheduling is still key in such adversarial environments.

5.5.2 Network deployment considerations

The two compute models discussed above, *CoOp* and *Ind*, needn't be homogeneously deployed in a distributed installation. In a networked deployment, a subset of OPaL nodes could be *CoOp* agents, training online, while most other nodes run *Ind* designs to meet throughput guarantees. The control plane would then combine, downsample, and distribute these improved policies between offline agents. This can be taken further still, using policy deltas or execution traces to enable out-of-path transfer learning for more complex models such as NNs.

5.6 Summary

We have seen through this chapter that online RL in PDP hardware is not only possible, but crucially offers tangible improvements to state-action latency and online learning throughput. This confirms one of the key assertions in my thesis statement, namely that 'In-network compute can make data-driven networking more efficient, effective, and responsive—enabling online learning to tailor policies to their deployment environment' (*S2*). The

key to doing so was to consider the architectural strengths and weaknesses of the target environment—in this case, SmartNIC devices—to creatively choose data formats and algorithms which suit their weaker, FPU-free, resource constrained designs rather than the state of the art. We have considered an off-path execution model, which places RL logic in-NIC, and have explored its essential role in preventing any impact to packet forwarding performance while enabling access to device-local state. By looking at these both, in tandem with the high degree of parallelism that SmartNIC devices engender, the *ParSa* algorithm was developed, which exploits the nature of tile-coded function approximation. Moreover, I've shown that it subdivides into neatly disaggregated tasks and is thus wait-free when combined with an atomic aggregation mechanism. We have examined two concrete implementations of OPaL—*Ind* and *CoOp*—which apply SmartNICs' parallelism in different ways to tailor state-action latency or online and offline throughput according to operators' needs. These are driven by effective methods for storing policy data across a non-uniform memory architecture, efficient internal and external communication, and careful task scheduling. OPaL was empirically evaluated on a number of benchmarks sized to large policies, confirming its reduced state-action latency and increased throughput compared to host-based execution. Observed performance numbers justified design choices in the scheduler, and validated the off-path execution model's ability to protect other cross-traffic carried by the NIC. Moreover, OPaL's runtime costs exist on a similar order of magnitude to existing PDP ML works implemented on the same hardware, indirectly confirming its validity and suggesting that sub ns execution might be offered by bespoke FPGA implementations. Finally, we have discussed how OPaL might combine with other state-of-the-art PDP works to realise the anti-DDoS agents presented in chapter 4 entirely in PDP hardware.

Chapter 6

Scalable Flow Classification

The ML techniques we have considered so far are powerful and effective and, with some amount of work, many can be ported to fit quite neatly into PDP hardware. Between the existing literature and the novel additions demonstrated until now, we have a toolkit of models and runtime techniques that neatly runs the gamut of DDN use cases' needs. Real-time analysis of operational Internet, WAN, and data centre traffic using granular, device-local state is that much more feasible because of their development. This can be through accurate flow characterisation (or classification), which can drive intrusion detection, prioritisation of traffic for certain customers, providing path-diversity, as well as marking the QoS of various users and protocols (Bernaille *et al.*, 2006; Roesch, 1999); some of these use cases are feasible even with the sampled, imprecise µs and ms-level data of sFlow, Netflow, and IPFIX (Aitken *et al.*, 2013; Claise, 2004). We have seen through section 2.4.1 the sorts of advances in dataplane monitoring which will allow us to further develop traffic analytics and classifiers, such as precise ns-level timestamping, INT, and flow-state analysis. For instance, problems such as microburst detection rely on extremely fine temporal or queue-specific properties visible *only* to PDP hardware. The catch is that the volume of per-packet or -flow data produced by such measurement imposes high packet-per-second and bandwidth constraints beyond the capabilities of host machines or even the routing fabric[1]—we *must* process it locally in the PDP environment. Even the most sophisticated software solutions process packets orders of magnitude slower than current backbone traffic of large operators, making them unusable for large-scale operational analysis (Park & Ahn, 2017).

Following the logic above (and throughout chapter 3), improvements to traffic classification and other DDN goals come from two directions: better data and more advanced ML techniques. While the community has come a long way in enabling in-network ML—and the in-situ processing that PDP-generated data can require—these nascent techniques still have their limits. Not all ML models are small enough to be expressed in these devices.

[1] Moving this data outside of PDP devices naturally occupies its own share of bandwidth. To transport it to a host machine, we must either scale up the capacity of the control plane to carry telemetry, or move it in-band via the dataplane (where it will add a proportional overhead to actual traffic). In the latter case, we *can* mark telemetry traffic as lower priority, but this would cause downstream collectors to make decisions on partial data due to losses (potentially having worse or less accurate outcomes).

Equally, some operational tasks rely on detecting spatial or temporal properties in data which can only be captured by more complex function approximators like CNNs, RNNs, or LSTMs. Barring the use of experimental architectures like *Taurus* (Swamy *et al.*, 2022), we have no mechanism to express these primitives in the dataplane.

What can be done to bridge this gap? So far we've also seen that PDP hardware excels at aggregating and fusing both measurement data and the intermediate results of distributed computations (sections 2.4.1, 2.4.2 and 2.4.4). It is evident that this line of thinking is what we need to help both the network and end-hosts scalably process telemetry data. This introduces its own set of challenges. In the case of timing information, we don't want to lose too much of the precision of individual measurements. The representation we choose must also preserve structural or temporal features pertinent to the traffic classes we detect.

[2] Pronounced "SAY-ther", Seiðr (a *cord*, *string*, or *snare*) is an Old Norse form of divination magic focussed on the reading and weaving of threads to know or alter fate. The admittedly tenuous connection is that flows are these threads to be read.

To solve these prior challenges, I present Seiðr[2], a dataplane-assisted flow classification solution. The design philosophy of Seiðr achieves the above goals: dataplane devices create accurately timestamped, aggregated histogram data structures for later analysis, while a scalable software stack performs more complex ML-based classification on commodity machines. As a concrete use-case, we look at fine temporal dynamics of TCP CCAs. Understanding and classifying them is important for network providers as inadequate choices have severe effects on transfer rates, especially in networks with a high bandwidth-delay product (Cardwell *et al.*, 2016) and in networks where multiple CCAs are used (Ware *et al.*, 2019). By using accurate congestion control diagnostics, operators will be able to infer sender problems (e.g., backlogged or application-limited senders), network inefficiencies (e.g., increased path latency and congestion), as well as receiver issues (e.g., delayed acknowledgements, small receiver windows) and fairness issues between delay-based and loss-based algorithms (Ware *et al.*, 2019).

The work presented in this chapter considers how PDP hardware can reduce fine-grained inputs and measurements into digests suitable for ML models running on host machines, and is based upon 'Seiðr: Dataplane Assisted Flow Classification Using ML' (K. A. Simpson, Cziva & Pezaros, 2020). I first consider and outline approaches and algorithms for generating and emitting histograms of packet- or flow-level statistics on PSA-compliant dataplanes (section 6.1). Then, in section 6.2, I examine a use case to which both histograms *and* precise packet timestamps are well-suited: the identification of TCP CCAs from the distributions of IATs. This is explained from observations of raw data and analysis of the BBR algorithm. Section 6.3 examines the performance, scalability, and effectiveness of the classification use case on a variety of ML models and Seiðr histograms. Finally, section 6.4 summarises the findings of this chapter.

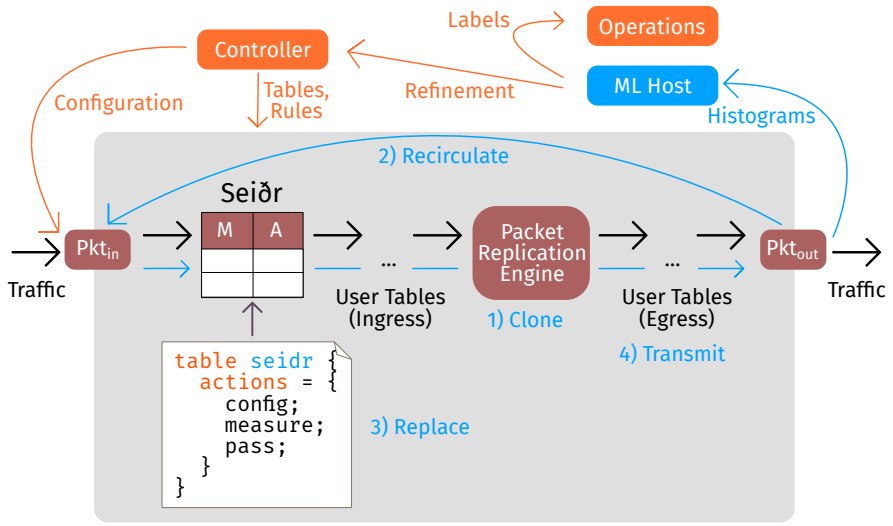


Figure 6.1: Seiðr's integration with a PSA-compatible dataplane, without the use of digest *externs*. Seiðr uses a register block to store many histograms, each comprised of a fixed bucket count. The main Seiðr table is used to select which flows or packets are tracked and used in the histogram process, allowing runtime control over monitoring via the control plane. When not using digests to emit telemetry (i.e., for transmission over the control plane), we must rewrite and modify cloned packets to contain the histograms—shown in blue.

6.1 Telemetry aggregation in the dataplane

Recalling much of the discussion in section 2.2.2, PDP hardware restricts the programming models and datatypes we may make use of, such as FPUs which would otherwise be helpful for the kinds of aggregation and statistics we're interested in. Chief among them though is their limited low-latency memory—$\mathcal{O}\left(10^1\,\text{MB}\right)$ (Jin *et al.*, 2017). In light of these restrictions, histograms become a natural choice. They are particularly suited for monitoring *distributional* characteristics of one or more features, an example of which will be introduced shortly. Each bucket is encoded as a fixed-size integer, and assuming we know the data ranges and granularity of traffic features we're interested in we can tune maximum bucket counts to fit the desired flow count into a given memory budget. At lower bits per bucket, we need only increase the histogram transmission frequency to compensate for numeric overflows. I describe here procedures for their creation and transmission under variations of the PSA dataplane.

6.1.1 Histogram generation

Although packet timing information is useful in understanding network and flow behaviour, without volume or packet rate reduction it's prohibitively expensive for hosts to handle each packet. Histogramming acts as the *aggregation step* which makes this class of analysis feasible in high-speed net-

```
header seidr_cfg_t {
  bit<8> function;
  bit<144> payload;
}
```

```
header seidr_t {
  bit<128> src_ip;
  bit<128> dst_ip;
  bit<16> src_port;
  bit<16> dst_port;
  bit<16> eth_type;
  bit<BUCKETS * 16> histo;
}
```

Figure 6.2: P4 headers for Seiðr configuration and histograms.

works. Figure 6.1 demonstrates how Seiðr, installed as an additional table in any P4 program, records and transmits inter-arrival time histograms. The format for these histogram packets is outlined in fig. 6.2; bucket counts and size are fixed at compile time. I choose here to store individual buckets as *u16*s, and fix the number of buckets to 100 per histogram as an example (and in later evaluation). Packets traverse a table which requires 3 actions to be implemented:

1. *config* reads any matched packets as a *seidr_cfg_t* of type *SET_*{*MIN*, *MAX*, *DST*, *SRC*, *LEN*} by using the P4 parser. These update registers 1–5 in table 6.1, dropping any matched packets.
2. *measure* calculates the inter-arrival time, update per-flow histograms, and transmits finished histograms to the correct host. I describe its operation in algorithm 4.
3. *pass* ignores packets, and is the default action.

Constructing Seiðr in this manner allows the control plane to install rules to enable or disable runtime reconfiguration as needed, and to monitor as many or as few flows as desired (i.e., using wildcard rules or exact matching).

Seiðr's operation—algorithm 4—is generally rather simple. For now, we will leave aside lines 5–12 until section 6.1.2. When a flow is to be measured, we first hash its 5-tuple (h) and determine which flow is currently occupying slot h (lines 1–3). In the event of hash collision (line 13), we ignore packets outside of the tracked flow to ensure that data is accurate. As later processing and classification directly affect what decisions are made by operators or automatically taken by a policy (possibly leading to incorrect flow limits or QoS choices), avoiding corruption and cross-contamination of operational data is paramount. If the slot is unoccupied or belongs to the current flow, we compute the IAT and assert ownership over the hash table slot (lines 14–15). Assuming the computed IAT is within bounds, we compute its bucket index in this interval and increment that bucket and a global counter (lines 17–19)—the global counter triggers a packet emission if it exceeds a known *Len* (line 20). Finally, we update the flow's last timestamp (line 22). To gain collision resistance, *Robin Hood* (Celis *et al.*, 1985) or *Cuckoo* (Pagh & Rodler, 2001) hashing could be used up to a maximum distance in the

Algorithm 4: Seiðr histogram update and transmission.

```
Data: 5-tuple, P4 metadata, P4 headers, Registers
h ← hash(5-tuple);
index ← BUCKETS * h;
owner ← HistoOwner[h];
if metadata.packet_path = RECIRCULATE then
    headers.tcp.valid ← false;
    headers.udp.valid ← true;
    headers.seidr.valid ← true;
    copy 5-tuple into headers.seidr;
    rewrite headers.ip, headers.udp using HistoSrc/Dest;
    headers.seidr.histo ← HistoData[index..];
    truncate payload;
    zero out registers: BucketCount, HistoOwner[h],
      HistoData[index..];
else if owner = 0 or owner = 5-tuple then
    HistoOwner[h] ← 5-tuple;
    iat ← LastTimestamp[h] - metadata.mac_ingress_time;
    if iat ≥ Min and iat ≤ Max then
        bucket ← BUCKETS * (iat - Min) / (Max - Min);
        HistoData[index + bucket] ← HistoData[index + bucket] + 1;
        BucketCount[h] ← BucketCount[h] + 1;
        if BucketCount[h] = Len then
            mark packet for cloning and recirculation;
    LastTimestamp[h] ← metadata.mac_ingress_time;
```

Table 6.1: Seiðr register map and required sizes using an an h-bit hash.

Field	Datatype	Count
Min	`u16`	1
Max	`u16`	1
Length	`u16`	1
HistoSrc	`u16 + u128`	1
HistoDest	`u16 + u128`	1
BucketCount	`u16`	2^h
LastTimestamp	`u64`	2^h
HistoOwner	`3 * u16 + 2 * u128`	2^h
HistoData	`BUCKETS * u16`	2^h

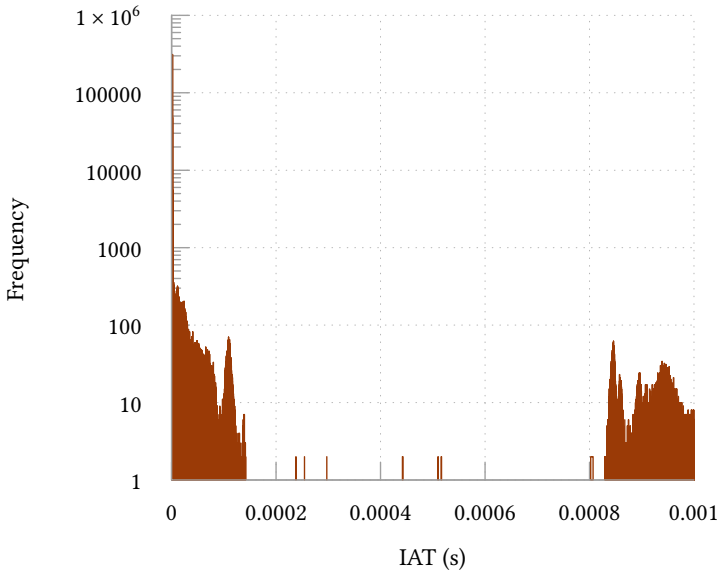
table, treating a zeroed owner as empty and an illegal source IP address as a tombstone value.

This design allows runtime configuration of all aspects save for the bucket count; at runtime, the only way to increase bucket resolution is to examine a smaller region of IATs. While in theory this could be configured below a maximum compiled into the firmware, the difficulties introduced by classification and later data processing make this infeasible. Unless using stream-capable classifiers such as LSTMs, changing the input size requires retraining from scratch since new neuron weights must be added and structural properties of the input data change. Increasing the bucket count requires new firmware installation, as many dataplane P4 implementations cannot allocate variable-length stores due to the lack of a dynamic allocator.

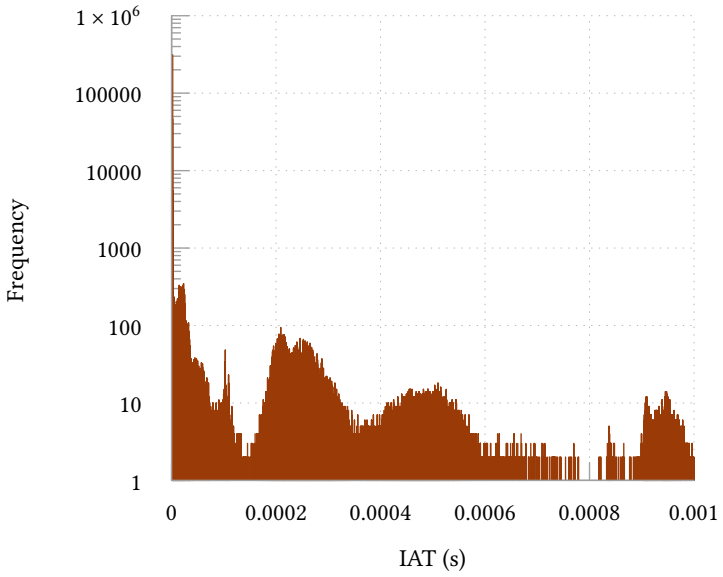
As a visual example of dataplane-generated histograms, fig. 6.3 shows the distribution of inter-arrival times between two TCP congestion control algorithms under otherwise identical conditions. While I cover the underlying causes for these stark differences in more detail later (section 6.2), it should be clear that there are variations in the *distribution* of features that are visible to us as humans quite plainly. It stands to reason that they should also be visible to a trained ML classifier. While these particular histograms are taken at the macro level—over the entire lifecycle of a flow—this also gives us cause to look into shorter measured windows.

6.1.2 Histogram transmission

When targeting the P4-PSA, we have some options in how histograms may be transmitted from the PDP hardware of interest—either to the control plane or dataplane. If we choose to emit histogram packets to the control plane, we may make use of *digest* `externs` which are offered by many PSA-compliant devices (though aren't strictly required). Practically speaking, these allow us to pack and emit arbitrary `structs` which are delivered to the attached controller CPU over the P4Runtime API.



(a) *TCP Cubic*



(b) *TCP BBR*

Figure 6.3: Example dataplane histograms showing visible differences in inter-arrival times of selected TCP flavours. These plots examine sub-1 ms dynamics of two separate flows of 1000 Mbit/s traffic generated according to section 6.3.1, over these flows' entire lifecycles. BBR flows have significantly more IATs measured in the 100–800 µs band, with additional structural differences outside this region.

Algorithm 4 and fig. 6.1 instead show a more complex case, where we emit generated histogram packets in-band on the dataplane. The PSA does not have any explicit mechanisms for generating new dataplane packets. To circumvent this, any packet which would complete a histogram is tagged for cloning at the end of the ingress pipeline, and recirculation at egress (line 21). This truncated copy returns to Seiðr's table, where we enable the relevant headers, change L2/3 fields, and write out the histogram contents (lines 5–12). The P4 deparser outputs the new protocol stack at egress, and transmits the histogram UDP packet into the network. Although achieved in a somewhat roundabout way, this does have an upside in that it is compatible with the guaranteed core functions of the PSA. In future, event-driven architecture proposals (Ibanez, Antichi *et al.*, 2019) may allow first-class support for packet generation.

There are tangible reasons to prefer emitting histograms to the dataplane in spite of the added complexity and recirculations. Although the data volume and packet-per-second reductions offered by Seiðr histograms are strong, the former case pushes the steering of all generated packets onto a single controller per switch. Intuitively, this is expensive and may very well interfere with regular operation of the control plane. On the other hand, outputting histograms over the dataplane allows administrators to make use of existing ASIC-backed infrastructure to load-balance classifications across typical host machines. Alternatively, we might forward packets directly to dedicated, network-connected accelerators like *BrainWave* (Fowers *et al.*, 2018) whose outputs then inform control plane operation.[3] Both of these cases spare the control plane of extra load, at modest bandwidth cost due to the reduction in telemetry volume. Digests of course have their own benefits: in some PSA implementations they may be built in the egress pipeline, adding flexibility for how Seiðr might be integrated with existing forwarding plane designs (rather than forcing its inclusion in the *ingress* pipeline).

[3] We can connect to such accelerators using a dedicated out-port, or by relying on the existing routing infrastructure if we need more flexibility.

Another design constraint where these two differ is in header-size limits.[4] In the case of digests we can assume that a P4 switch is capable of emitting larger packets: following the specification, switches are free to collate generated digests together before handing them off to the control plane. For transit via the dataplane, however, platform-dependent limits will apply to output packets—particularly as we must use rewritten header fields for this purpose. This leads to a problem which algorithm 4 does not directly account for: histogram data may be larger than headers allow for. To fix this, we must make some high-level modifications to its behaviour. We must mark in-progress transmissions and emit bitslices from the output histogram split over several egress packets by performing a clone loop, where each non-terminal histogram packet is cloned and marches the 'send window' forward. Until termination, updates to the histogram's recorded data are blocked. A key limitation, however, is that the histogram generation frequency and other parameters must be tuned to prevent traffic amplification driven by the switch.

[4] These designs and implementations again focus on using Netronome NFP SmartNICs, as opposed to Tofino RMT hardware where such limits do in fact apply.

6.1.3 Accurate, precise and high-resolution timestamping

Precise timestamps are critical when detecting temporal properties of flow behaviour, such as microbursts or inferring flow CCAs. It is especially important in high speed (100 Gbit/s) networks, where there can be as little as 6.7 ns between packets that need to be analysed. With a Linux-based software solution (e.g., reading packets from a link with *tcpdump*), the Linux kernel can only provide microsecond-level accuracy with precision in the order of 100 µs (Kundel *et al.*, 2020). DPDK improves on this, increasing the accuracy to 100 ns in the best case (Primorac *et al.*, 2017). However, today's dataplane devices (e.g., Netronome SmartNICs, NetFPGA SUME) allow nanosecond-accurate timestamps to be retrieved from the MAC modules with a precision of 10 ns (Kundel *et al.*, 2020), a timestamp property Seiðr relies upon.

6.2 TCP congestion control classification

Figure 6.3 suggests that a notable use case for this type of measurement is detecting TCP flows' choice of CCA. In a TCP connection, each machine is free to choose the CCA it uses to send bytes, and thus how it responds to network congestion signals. This choice is entirely local, and so is invisible to both the other machine and the network. In data centre networks, operators choose these ahead of time to ensure optimal behaviour, where the environment makes it easy to consistently deploy this choice across all nodes. This is not the case in most ASes; in a transit network or large WAN, these hosts (and thus the CCAs in use) are outside the control of network operators, which introduces difficulties when CCA interactions lead to *unfairness*. Consider the recent (and widespread) introduction of *TCP BBR* (Cardwell *et al.*, 2016). *BBR* is a delay/model-based CCA which converges on a fair share of bottleneck bandwidth by reducing its rate if the round-trip time increases, while periodically attempting to increase send rate to account for path or load changes. However, *BBR* traffic can consume 40 % of link capacity when multiplexed with loss-based CCAs, regardless of the number of competing flows (Ware *et al.*, 2019). When ensuring fair transit to all flows, this is hardly a desirable outcome; in fact, it's one which may frustrate clients or violate SLAs.

A curious property of *BBR*'s algorithm which sets it apart from other variants is that packet transmission is *timer-based*. `send(packet)`, as defined in the canonical algorithm, asks that on transmission of a packet, the sender should wait for the estimated time that packet would take to reach the recipient. For instance, at an estimated bottleneck bandwidth of 8 Mbit/s, a 1024 kB packet would hold back the next packet in the flow until 976.6 µs had elapsed. When packet sizes remain similar this causes strongly periodic behaviour, while mode switches in the *BBR* algorithm cause these periodic

bands to shift up or down accordingly. This effect is stronger than in existing loss- and delay-based algorithms which remain intrinsically tied to the notion of a congestion window (where release of buffered packets follows the receipt of *ACK* messages). As a result, timing behaviour of past CCAs may be influenced by (the lack of) packet pacing, periodic components might be made noisier by jitter along the return path, or the behaviour of the receiver might add further noise.

This high-level analysis of *BBR* gives us a strong feature to use as the basis for classification: the IATs for each packet in a flow. We have two options for processing this for classification: we may use a compressed, fixed-size representation such as histograms to capture the aggregate distribution, or we may attempt to capture structural behaviour by using a variable-length stream of IATs. In many networks, the data and packet rate reduction offered by the former is required to make this possible. Indeed, as we've examined in greater detail through section 2.4.4, in-switch aggregation has seen great success in aiding ML for training (Y. Li, Liu *et al.*, 2019). We make use of the following standard classification algorithms on a fixed-size representation to attempt to single out the CCA in use:

- *k-Nearest Neighbours* (*k*NN). A simple and well-understood classifier which assigns labels based on the closest members of the training corpus (i.e., by the ℓ_2 metric). They have a linear runtime memory cost in amount of training data, and no training cost other than loading all data points. However, they are surprisingly capable of learning complex decision boundaries on fixed-length inputs.

- *Convolutional Neural Networks* (CNNs). As discussed in section 3.2.2, CNNs are a neural network approach which learns convolution kernels to classify fixed-length data, particularly when recognising spatial features. Runtime memory cost is fixed for a given architecture irrespective of the amount of training data used, with a high training cost in memory and computation time.

When examining *k*NN classifiers, I measured accuracy across choices of $k \in [2, 8]$; I found $k = 2$ to be the most effective choice with these input data using the ℓ_2 metric. Our CNN architecture is described in table 6.2, using ReLu activation and 1×1 stride in convolutional layers unless stated otherwise. Training occurred over 5 epochs using the Adam optimiser with categorical cross-entropy as a loss metric, and a batch size of 64 histograms (8 for full sequences due to the smaller data volume). For *BBR vs. Cubic*, the complete model consists of 104 898 × 32 bit floating-point parameters (409.76 KiB), while the full classification task adds a further 130 parameters (0.51 KiB).

Table 6.2: CNN architecture for 100-entry histograms.

Layer	Nodes/Filters	Filter Size	Output Dimension
Conv2D	32	3 × 1	98 × 1 × 32
MaxPool	—	2 × 1	49 × 1 × 32
Conv2D	64	3 × 1	47 × 1 × 64
MaxPool	—	2 × 1	23 × 1 × 64
Conv2D	64	3 × 1	21 × 1 × 64
Flatten	—	—	1344
Dense	64	—	64
Dense (Softmax)	$n_{classes}$	—	$n_{classes}$

6.3 Evaluation

The performance of Seiðr is evaluated from several angles. On classification, we are interested in the accuracy of CCA detection using IAT histograms, the time taken to train a model, and the required time to classify a flow. In the *2-class* problem, we investigate whether it is possible to separate TCP *BBR* from *Cubic* using IAT histograms as the input data, while in the *4-class* problem we extend this to include *Reno* and *Vegas*. I compare this work against Hagos *et al.* (2018) in this regard. On deployment, I demonstrate the bandwidth and memory requirements imposed by Seiðr.

6.3.1 Datasets

We examine synthetic flows modelling bulk data transfer at various speeds, generated using iPerf3 (Guéant, 2020), and processed using custom P4 firmware for Netronome NFP SmartNICs. Packet captures and IAT streams are publicly available (ESnet, 2019; K. A. Simpson *et al.*, 2019). For every pairwise interaction between TCP *BBR*, *Cubic*, *Reno*, and *Vegas*, we capture solo and multiplexed dynamics by running each flow for 3 s, with 2 s of overlap (i.e., the second flow begins at $t = 1$ s). I observed that the first flow always completes slow-start before multiplexing begins, and by construction we have several unimpeded captures for every flavour. The number or volume of multiplexed flows isn't expected to substantially alter captured dynamics (i.e., 3 flows at 300 Mbit/s and 2 flows at 200 Mbit/s should both have flows fall to 100 Mbit/s). Flows in one capture are generated using the same target rate in {100, 200, …, 1000} Mbit/s, each uniformly randomly perturbed within ±10 %. We also control how this rate limit is applied: *wire-limited* traffic uses `tc` in the Linux kernel to apply rate-limiting, while *application-limited* traffic uses iPerf's built-in mechanisms to control send rate. Application-limited traffic leads to specific behaviour in *BBR* and some other flavours, while wire-limited traffic creates loss events as the rate grows too high (which can expose additional behaviours in response to such events). 10 such captures are recorded for each (CCA_1, CCA_2, *speed*, *limiter*) tuple, and generated

flows are labelled accordingly.

IAT streams are broken down into overlapping sequences of the required length, before being histogrammed as required into 100 buckets over 0–1 ms. The use of overlapping sequences extends the training and testing sets significantly, while ensuring that larger sequences don't result in a far smaller training corpus. Cross-validation occurs on a per-flow basis rather than per-sequence, i.e., sequences from the same flow must only appear in *either* the test or training set. This ensures stringent data hygiene, and prevents adjacent sequences from inducing overfitting. All classifier evaluation which follows uses 4-fold cross-validation. The data is comprised of 4994 flows (832 in *2-class*), or 18–31 million sequences (3.2–5.2 million in *2-class*).

6.3.2 Experimental setup

All experiments were executed on a single machine running Ubuntu 18.04.4 LTS (GNU/Linux 4.15.0-96-generic x86_64), using an Intel Core i7-6700K CPU (4 × 4.2 GHz) which had 32 GiB of RAM. CNN training was performed using Nvidia RTX 2080Ti cards (11 GB GDDR6 *Video RAM* (VRAM)). For the dataplane, we used multiple Netronome Agilio CX 2 × 40GbE SmartNICs using 40 Gbit/s connections between source and destination hosts.

6.3.3 Classification performance

In the *2-class* formulation, we can observe from fig. 6.4 that CNN performance increases slightly with the length of the input sequence for classifying application-limited traffic. CNN-based detection has a peak F1-score of 0.965 for application-limited traffic, and 0.894 when wire-limited. This increase does not extend towards histograms taken over the entirety of each flow (*Full*), which are hampered by having 6 orders of magnitude fewer training samples. While very effective, *k*NNs come with significant memory cost. By design, the entire dataset must be kept in memory: for subflow histograms of length 500 packets, this equates to 1.5 GiB of training data. Naturally, this is undesirable for many network deployments, where easy relocation of inference may be key.

Figure 6.5 shows in the *4-class* case that we observe a sharp loss in classification accuracy, peaking at (59.5 ± 2.0) % for CNNs and (64.5 ± 1.6) % for *k*NNs. This suggests that IAT histograms don't generalise as an effective feature for other TCP flavours. Exploratory work with LSTMs on IAT streams confirmed that this persists before aggregation. Likewise, exclusive pairwise training did not lead to an increase in accuracy. However, fig. 6.6 shows that timing information remains key in separating BBR from its predecessors to a high degree of accuracy, confirming our hypothesis that its *timer*-based (rather than *cwnd*-based) design allows for this detection. If this marker were present between *loss*- and *delay*-based variants, then we'd also see

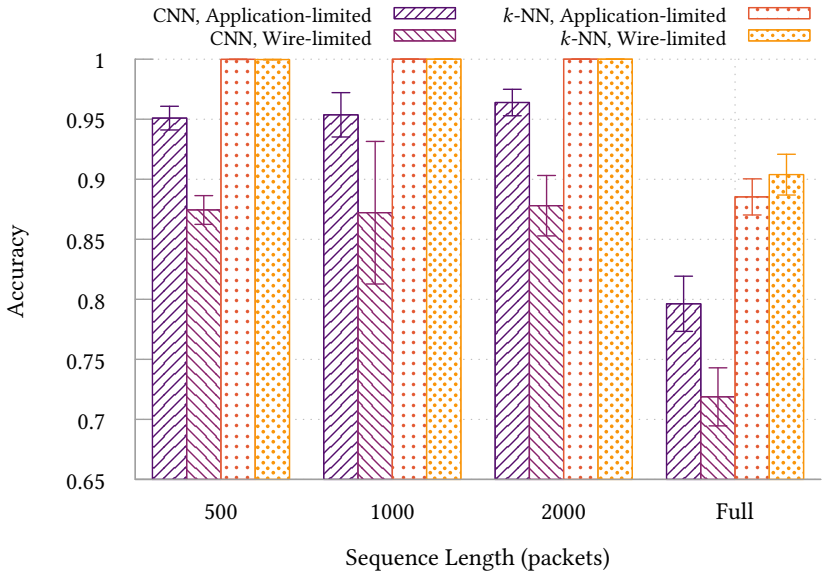


Figure 6.4: Accuracy of *k*NN and CNN classifiers when classifying *BBR* and *Cubic* TCP traffic from IAT histograms, trained over various sequence lengths. In both subsequences and complete flow histograms, accuracies are generally high (at least 87 % and 72 % respectively). *k*NNs outperform the CNN architecture here, otherwise we generally see that longer subsequences offer some improvement to application-limited accuracy, and wire-limited *Cubic* and *BBR* are harder to tell apart.

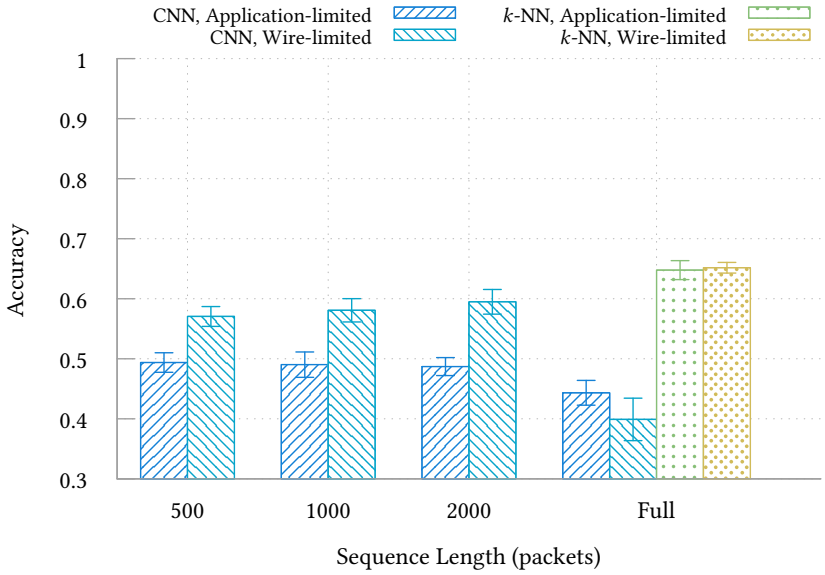


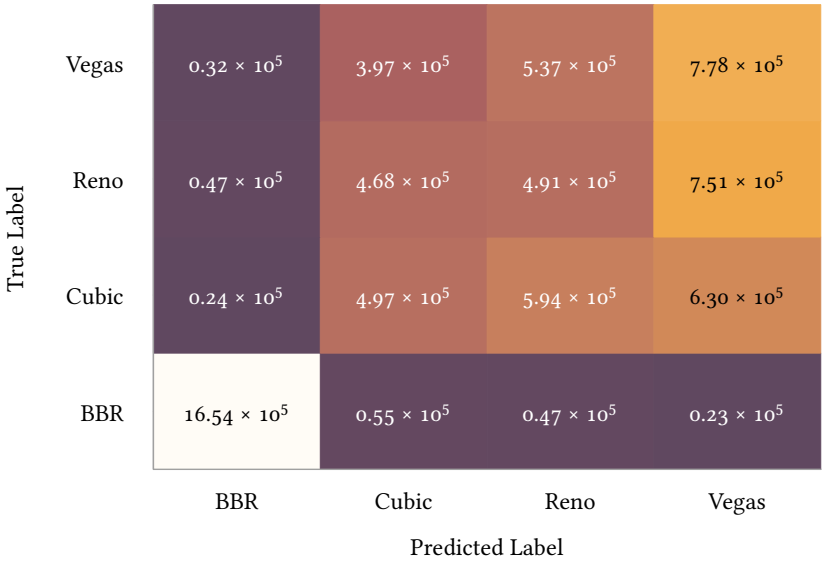
Figure 6.5: Accuracy of *k*NN and CNN classifiers when classifying *BBR*, *Cubic*, *Reno*, and *Vegas* TCP traffic from IAT histograms, trained and tested on various sequence lengths. *k*NNs again achieve better performance here, but when handling subflows the entire corpus cannot fit into RAM. We see instead that wire-limited traffic is easier to classify when all 4 CCAs are examined: again, this performance increases with histogram sequence length.

Table 6.3: Training, inference, and runtime memory costs of CNN and *k*NN models.

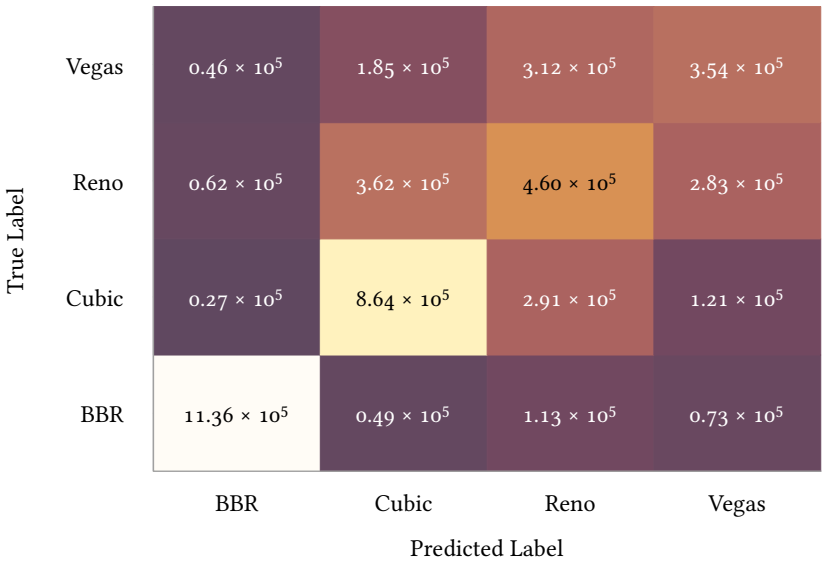
Family	Online (Subflow)	$n_{classes}$	Train	Test	Memory
CNN	✓	2	43(2) min	49.1(92) µs	409.76 KiB
	✓	4	243(2) min	50.5(17) µs	410.27 KiB
	✗	2	1.82(47) s	161.3(39) µs	409.76 KiB
	✗	4	7.94(50) s	137.7(12) µs	410.27 KiB
*k*NN	✓	2	21.4(12) min	323(69) µs	2.1 GiB
	✓	4	—	—	12.58 GiB
	✗	2	0.200(6) s	54.0(3) µs	332.8 KiB
	✗	4	2.20(4) s	517.0(50) µs	2.0 MiB

high predictive power over *Vegas* traffic. Breaking down these confusion matrices by rate limit type sheds still more light. In fig. 6.6a, application-limited data transfers are almost indistinguishable using these metrics (aside from *Vegas*), while fig. 6.6b reveals that IATs hold some discriminative power for wire-limited Cubic traffic. Note that *4-class* *k*NN experiments on all but full sequences required excessive memory and classification time, and so are excluded. While full-sequence *k*NNs outperform all examined CNNs on this task (respective peak F1-scores 0.697 vs. 0.486), these reduce $F1_{BBR}$ from 0.935 to 0.810.

I contrast this work with that of Hagos *et al.* (2018), who employ CNNs to predict *cwnd* size for any flow from its stream of bytes-in-flight measurements. On detection of a loss event the *multiplicative decrease* β is measured from estimated *cwnd*s, from which the CCA may be classified. In identifying TCP *BIC*, *Cubic*, and *Reno*, they achieve 95 % accuracy, which outperforms Seiðr on *cwnd*-based CCAs. Yet their approach cannot work for detecting *BBR*. *BBR* is not based upon the notion of a sliding congestion window, so there is no parameter β to infer. Although IAT histograms are suitable for *BBR* detection due to the intrinsic properties of its algorithm, we envision that this approach could be augmented by using a negative *BBR* classification to trigger *cwnd* estimation. Having seen that some predictive power from IATs is preserved for *cwnd*-based CCAs, we expect that this will increase the accuracy of a universal classifier. It is important, however, that this step be taken adaptively; this incurs higher resource requirements for bytes-in-flight tracking and for efficient handling of potential return-path asymmetry. Seiðr on its own does not add such overheads or operational complexity, and does not require a telemetry system to see or detect *cwnd* adjustments by the sender.



(a) Application-limited. $F1_{BBR} = 0.935$, $F1 = 0.486$.



(b) Wire-limited. $F1_{BBR} = 0.893$, $F1 = 0.573$. *Notably,* $F1_{Cubic} = 0.625$.

Figure 6.6: Confusion matrices for a CNN on the *4-class* problem, 2000-packet length sequences. Brighter entries along the diagonal indicate correct classifications. BBR remains easy to distinguish regardless of the rate limit mechanism, while Cubic is slightly more distinct from its *cwnd*-based alternatives for wire-limited traffic. CCAs are generally easy to tell apart in wire-limited traffic—this is sensible, in that most CCA differences will arise in response to congestion events rather than application behaviour, and matches the general trends seen in fig. 6.5.

6.3.4 Training and inference costs

Typical test and training times for these ML classifiers and problem formulations are lsited in table 6.3. Training times for *k*NNs include the time taken to load and process the entire training set, and are incurred *every time* the model is started on a new host. CNNs trained for online analysis (flow subsequences) achieve the lowest per-flow inference times, and are increased during offline analysis due to worse batching and cache behaviour on the smaller data set. While *k*NNs are effective in many cases, I find they are only computationally viable when offline (i.e., full-flow histograms), as the entire test data corpus must remain in memory. A single *4-class* cross-validation fold (2000 packets) required 3 days to train and test over the entire dataset, which was deemed to be outright infeasible. In contrast while online CNNs take longer to train, they have a considerably lower memory footprint, the training cost is paid only once, and flows may be classified in real-time with milliseconds of total observations.

6.3.5 Switch resource usage

The implementation of Seiðr requires an extra table in the ingress pipeline to update buckets, update configuration, and rewrite packets. If digests are used rather then clone-based packet rewriting, then this table may be placed in either ingress or egress. Further code space is required to include a configuration packet parser. Shared configuration data (registers 1–5) requires 42 B per switch, while each flow requires 224 B and 248 B to store buckets, counters, previous timestamps, and active 5-tuples on IPv4/v6 networks respectively. On platforms which support hash-table structures, this cost scales linearly with the number of tracked flows. Otherwise, this requires pre-allocation of an entry for every possible hash value (e.g., 14–15.5 MiB for a 16 bit hash). This small memory requirement fits histogram generation to all devices available today.

6.3.6 Quantifying in-network data aggregation

To show data rate reductions, I compute the compression ratio of generated histograms against various other representations which can be used to move IATs (or the packets used to compute them) from the dataplane. Although it is more commonly a metric used for compression algorithms, it is simply:

$$CompressionRatio = \frac{UncompressedSize}{CompressedSize}$$

Figure 6.7 demonstrates the reduction in data sent from raw mirrored packets, to a stream of measured timestamps or IATs, to Seiðr histograms on an IPv6 network. Timing histograms naturally provide a larger data reduction as the amount of measured packets increases, while a per-packet IAT or

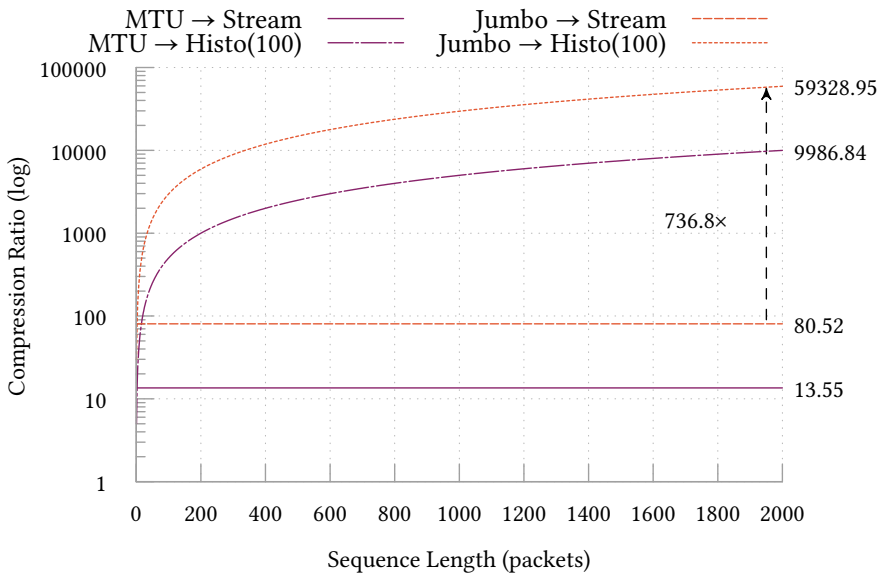


Figure 6.7: Compression ratio of 100-bucket histograms and timestamp streams from raw packets on an IPv6 network. As sequence length increases, histograms provide more of an advantage in compression rate, being 736.8 × smaller than timestamp streams when analysing 2000-packet sequences.

telemetry stream offers no reduction in packet rate. Due to this, 100-bucket histograms cause a greater data reduction than per-packet IATs after just 4 packets in a sequence, and consume 736.8 × less volume for 2000-packet sequences.

To make this concrete, 100 Gbit/s traffic is reduced to 10.01 Mbit/s additional switch traffic for MTU-size packets, and to 1.69 Mbit/s for jumbo frames. IAT streams, by comparison, reduce to 7.38 Gbit/s (resp. 1.24 Gbit/s). For a flow at 100 Mbit/s, only 30 ms is needed to collect enough packets to make a classification. Scaling beyond this, packet processing rates are the bottleneck. As commodity machines and today's stream processors have a reasonable upper bound of ~1 Mpps processing capacity (Gupta *et al.*, 2018), Seiðr could scale up to 1 Tbit/s MTU-size packet traffic on one machine, which would correspond to only 333 kpps histogram packets (55.6 kpps if jumbo-size). Reliably scaling to 10 Tbit/s and beyond requires only that we increase the histogram sequence length to ≥7000 packets.

6.4 Summary

Through this chapter, we've considered how PDP hardware can reduce fine-grained inputs and measurements into digests suitable for ML models running on host machines, supporting one of this thesis's claims: 'dataplane programmability will allow the precise measurement *and* data aggregation that can enable fine-grained data-driven analyses to scale to high flow rates

or large networks' (*s3*). In particular, I've shown PSA-compliant ways to implement in-network data aggregation in the form of histograms, tailored towards tracking ns-precise timestamps. Histograms of per-flow packet IATs have been presented as the input for various ML algorithms, including CNNs and *k*NN classifiers. We have seen empirically that Seiðr can successfully tell apart TCP CCAs, in particular, it identifies BBR from its predecessors with over 88–96 % accuracy, while only consuming a maximum 15.5 MiB of dataplane memory. We presented the trade-offs between training and inference times, memory requirements, and accuracy in the context of CNN and *k*NN classifiers and shown that Seiðr outperforms prior work by increasing classification accuracy on novel TCP CCAs, providing the ability to classify at very high traffic rates. Furthermore, we have identified a key temporal property of TCP BBR which allows its easy detection among other flows.

Chapter 7

Conclusion

This thesis has demonstrated how the advanced capabilities of modern PDP hardware can be used to make online learning—particularly classical RL—and other DDN use cases feasible in the dataplane. These capabilities move us closer to automatically inferring complex, tailored dynamics for network control and optimisation, in place of laboriously hand-tuned heuristics for the same tasks. Moreover, the network cooperation enabled by PDP hardware allows us to make these decisions on more precise data and to take more involved actions on packets and flows. In spite of how far this hardware has come, this has required many design caveats—we have examined various ML and RL primitives alongside the architectures of modern PDP hardware to pick out the considerations of running in these resource-limited yet high-speed environments. At the same time, I have also argued the case for the unity of DDN and dataplane programmability: by mitigating amplification DDoS attacks using an online multi-agent RL system, and by showing one way in which PDP hardware can support data-driven classifications for network management. Through these advances and thorough review of the literature, I have given substantial evidence for the value and viability of PDP networks empowered by data-driven methods, satisfying claims *s0–3*. Recalling the initial thesis statement:

> Data-driven networking—enhancing networks with ML—and dataplane programmability are key tools in aiding the control and measurement of future networks (*s0*). Data-driven methods such as reinforcement learning can lead to improved performance in network optimisation and control problems, such as DDoS prevention (*s1*). In-network compute can make data-driven networking more efficient, effective, and responsive—enabling online learning to tailor policies to their deployment environment (*s2*). Finally, dataplane programmability will allow the precise measurement *and* data aggregation that can enable fine-grained data-driven analyses to scale to high flow rates or large networks (*s3*). Applied together, programmable

> data-driven networks can improve computer network operation beyond the sum of these parts.

The value of each of these tools on their own for network management (*s0*) has been shown mostly by the use cases considered throughout chapters 2 and 3, but is also implicitly shown through the novel techniques presented in the thesis's main developments. RL's value in particular was shown by its use to learn to mitigate DDoS attacks in an online way (*s1*). By adapting classical RL policy formats and algorithm choices to suit the design, threading model, and FUs of manycore SmartNIC hardware, online in-NIC RL was made possible. Eliminating PCIe and host stack overheads offered substantial latency benefits, and the parallelised Sarsa algorithm brought higher-throughput online learning (*s2*). In-network aggregation of per-packet statistics to histograms made the handling of high-rate timestamps feasible, and enabled a flow classification task with clear operational benefits (*s3*).

What is far more interesting, however, are the wider takeaways and lessons learnt in the development of this work, and by collecting together a wide family of solutions falling under the DDN or PDP umbrella. Each has its own impact on the design and deployment of the other. Equally, it's worth mulling over the horizon of networking capabilities and form factors in the short and long term.

7.1 The need for co-design

Making use of DDN and PDP-accelerated solutions is, as this thesis has likely demonstrated, an involved process. In the DDN case, 'zero-touch' deployment and development are likely impossible. While we can train successful policies, DDN cannot itself derive the *mechanisms* of control: action models, reward functions, and the state which they should operate on. Learnt policies and parameters operate as well as they can *within the framework we give them*, and generally succeed at so doing. Yet as we've seen already, by designing DDN solutions without deep, cross-disciplinary human expertise on the controlled system we can easily introduce catastrophic impact in critical scenarios. This extends even to testing and training environments; capturing every real interaction is crucial if one has any hope of generalising to production networks. At the compute scales of interest, namely small and fast models with lower sample complexity, factoring in human expertise ahead of time can be useful in accelerating inference and training as opposed to completely 'clean-slate' approaches. There is an argument to be made about to what degree we *should* be integrating our own intuition—biasing models away from potentially better solutions, i.e., that reward is all you need (Silver *et al.*, 2021)—but we must often perform our own feature engineering regardless. Temporal properties of state are one such example: we *could* make use of LSTMs and similar constructs to cap-

ture them automatically, but the price paid in complexity is less than appealing.

Easily taking advantage of PDP hardware requires less thought. Automatic offload tools are already very promising for (mostly) cutting host machines out of the packet processing loop, and extracting data and pipeline parallelism when they cannot. I expect this tooling will only improve further. For novel in-network compute uses or latency-optimal solutions, we should not expect any automatic wins. More than anywhere else, these demand dedicated co-design, deep integration with hardware-specific communication primitives, and retooling to account for parallelism and heterogeneous compute models. Today's—and likely tomorrow's—diversity of device designs and programming models is a blessing and a curse. It is necessary that we have such variation to achieve a balance of performance, price, and capability across market silicon. The downside is that this lack of unity demands insight and expertise on the devices themselves, rather than a single network-compute model, but making best use of them is intensely rewarding. This is no surprise—taking advantage of parallelism alone is difficult (Sutter, 2005), but is itself necessary nowadays. In combinatorics for instance, automated fork-join frameworks produced subpar results, necessitating similarly "intrusive" co-design to OPaL which "[increased] the amount of code needed by as much as an order of magnitude" (McCreesh, 2017, p. 214). SmartNICs and PDP hardware impose a stronger blowup in complexity. The intrusiveness extends in our case beyond parallelism: to the memory model, to specialised capabilities like TCAM-backed MATs, and to missing FUs we might otherwise have taken for granted such as FPUs. Many of the required data structure transformations cannot be automatically derived, and the best algorithms for in-network compute take advantage of tightly coupling these elements together. Weighing these against one another requires real trade-offs to be consciously made at all levels of our design—one might also argue that this makes in-network solutions fragile against future hardware. This is not to say however that we won't (or shouldn't) have further developments here. The ingenuity of engineers, novel PDP architectures, and economic drivers in data centre-scale applications will see to it that this is a long-lived wellspring of research. It is simply the case that, as in parallel computing, there is no free lunch (Sutter, 2005).

7.2 A challenging security context

At present, ML and DNNs have a wide variety of viable attacks at runtime and during training, which raises questions about whether their use in DDN is safe—either online or offline. Attacks and defences still appear to be rapidly iterating against one another, and as such it is not clear that we should be focussing on integrating specific defences while they appear to have such a short life-time. I argue that DDN deployment can be safe so

long as there is a reasonable degree of isolation between a hypothetical attacker (or self-serving client) and the model. The meaning of 'isolation' in this context changes depending on the attacks we want to defend our DDN system against. For instance, destructive steps in processing, true isolation as in many resource placement problems (i.e., DDN applied at design-time of a network or circuit), or aggregation of input data may aid against evasion, poisoning, and adversarial behaviour. The network itself offers some degree of isolation of many outputs, which might make model stealing and evasion more difficult. Transient network conditions make flow statistics noisier, pushing them away from the intended perturbation, while routing and QoS decisions might only be inferred indirectly with added noise and delay. In closed loop circumstances, isolation is less clear. This is purely intuitive reasoning—in the longer term, we require further research tailored to the network problem-space. Future studies should aim to offer a set of quantitative bounds on how an attacker's input can affect learnt models in reasonable scenarios, measured specifically on network tasks. These must account for the effects of data aggregation, processing, and noise from cross-traffic—as well as path characteristics introduced by other ASes and network segments en-route to a target.

7.3 Future directions

Given that one of the advantages of RL methods is their ability to dynamically learn by trading off exploration and exploitation, precisely how well-suited they are to handling the evolution of networks is an interesting research question. Handling non-stationary problems is *possible*, but rarely recommended, particularly with DNN-based policies. To respond to such change, we simply need to either scale up gradient contributions, or increase the strength of exploration parameters like ϵ. Yet we are left with two key challenges. The first is that we need robust means of detecting the kinds of problem-space evolution we're interested in. There are tricky tuning factors to consider: chief among them are handling seasonality, and the timescales and magnitudes of evolution worth adapting to. If traffic varies diurnally, for instance, then choosing the wrong timescales would likely cause an online learner to oscillate between policies—meanwhile, the desired behaviour would be to learn to handle these modes in the *same policy*. The techniques discussed in section 3.3.5 may be useful to this end, such as adaptive exploration, changepoint detection, or signal processing methods (whose intersection with RL seems as yet unexplored). The second challenge is that we must understand and model what problem-space evolution *really* looks like. While it is known that DDoS attack strategies evolve in real time (Kang *et al.*, 2016a), to my knowledge no works detail what patterns such evolution might take. In the wider Internet, aggregate changes in bandwidth and usage are likely easy enough to model (Bauer *et al.*, 2021). But, barring historic case studies, estimating the effects of new protocols

and CCAs before their deployment is unlikely to be feasible.

Online RL via OPaL is limited to devices in a SmartNIC or NPU-style form-factor. This is less than ideal for larger deployments, yet achieving this level of flexibility at switch scale is unlikely to be possible without heavy concessions. Register access limits and a fixed number of pipeline stages would make a purely P4-PDP variant difficult to express, let alone a MAT-accelerated approach (which would be dependent on the controller for policy updates). As discussed in section 3.2.1, we may be able to use MATs to perform or accelerate the tile-coding step, but accessing and later updating action values stored in registers is likely to be incompatible with RMT designs. A promising avenue here would be to investigate ongoing transfer learning between online OPaL agents and high-throughput offline function approximators such as BNNs. This might allow, for instance, having a canonical 'known good' policy in the majority of the network installed to Tofino switches, while a smaller proportion of flows or packets are routed through actively learning bump-in-the-wire nodes. The control plane is then responsible for collating their local policy modifications and generating a set of BNN parameters which expresses the same decision boundaries as the aggregated tile-coded policy.

While this thesis achieves online RL in PDP hardware, it does so by choosing a function approximation scheme with lower model capacity than more common alternatives such as NNs. How could we enable online learning for these more complex approximators? Practically speaking, minibatches and replay buffers are necessities and will require storage in high-speed memory. This is somewhat counter to the design of PDP hardware, but it wouldn't be too onerous a requirement in bespoke designs. Computing gradients themselves in a way which is scalable and tailored to the execution model (many weaker cores or FUs) remains a challenge. We might find value in combining insights from the field of distributed model-training, such as wait-free backpropagation, to achieve low-latency forward passes and parallelised updates to the policy when using model-parallel inference. Here though, we would constrain the scope of such algorithms to a single device, which might enable some shortcuts and further optimisations. This continues to make use of the many cores or FUs that we might expect on SmartNICs or FPGA devices—*N3IC* (Siracusano *et al.*, 2020) offers a model-parallel NFP implementation of the NN forward pass which might be compatible in theory. BNNs are not, however, suited for this purpose, given that training requires incremental real-valued adjustments to the model parameters. As such, online NN training in the PDP would likely mandate *at least* fixed-point arithmetic, ruling out the strong performance benefits of BNNs. ES methods may be an exciting avenue here—devoid of *any* gradient computation, they instead add uniform noise over the entire parameter set $\boldsymbol{\theta}$, making it computationally cheaper to train a policy than the use of backpropagation.

Future networks and PDP hardware designs will become only more varied

and vibrant as time goes on. Hearkening back to my earlier thoughts in section 2.5, the hardware advances we've examined the wider impact of are only the first wave of truly programmable solutions. I expect that we will see many more points along the capacity-capability Pareto front—blurring the lines between compute classes as CPU-NIC co-designs are beginning to do. Currently, the solution is to mix and match devices as a hybrid SoC board (as in Intel's IPUs), but we should hope for a radical shake up in much the same vein as RMT sooner or later. Perhaps this will tear down some of the roadblocks which make online (and otherwise in-PDP) ML difficult today—such as on-device state modification, or even piece-wise replacement of tables and logic (e.g., encoded as smaller P4 or eBPF subprograms). Beyond hardware-based dataplane programmability, the rearchitecture and accelerated packet processing stacks we've seen from OSes in the last few years alone bode well for software dataplanes. While there will of course be iterative improvements to XDP and similar frameworks to make them more capable, user-friendly, or better exposed to applications, more specialised kernel and network stack designs will likely arise. After all, the commodity CPUs which will inevitably be co-hosted with our SmartNICs and FPGAs will need an accelerated and predictable stack for processing packets and flows. In time, I expect that while standard (non-PDP) routing hardware will still make up much of the Internet's backbone and capacity, if PDP becomes lower cost and more energy-efficient then we will see greater proliferation of programmable network infrastructure from the core to the edge. It may be the case that we'll never reach the original *active networking* vision of a fully cooperative and user-controlled routing fabric—perhaps this shouldn't be the case—but I think that this will enable a new kind of evolution in vendors, ISPs, and host networking stacks. Networking, as a field, is on the cusp of some truly interesting developments—and I'm excited to see where it all leads.

Appendix A

Protocol Trends in CAIDA Traces

In an earlier version of the work in chapter 4 which we chose to submit to the EuroS&P '19 conference, I backed up my analysis of the shortcomings of *Marl* (Malialis & Kudenko, 2015)—i.e., that TCP traffic is both dominant and negatively impacted—by referencing a study from M. Zhang *et al.* (2009) which suggested that TCP traffic was most prevalent in packet and byte counts for Internet traffic. In addition to other feedback mostly unactionable short of working for a hyper-giant/-scaler operator, Reviewer 1 raised the challenge that:

> The statistics borrowed from [M. Zhang *et al.*, 2009] are 10 years old by now; I imagine that the Internet traffic has changed significantly by then due to streaming services and new protocol developments.

So-challenged—in spite of the fact that streaming video is most often DASH-based (i.e., carried over HTTP, so TCP) and that prominent new protocols are themselves congestion-aware (e.g., QUIC)[1]—I carried out a high-level analysis of the CAIDA 2018 passive traces dataset (CAIDA, 2018) while the school's network infrastructure was otherwise knocked out due to a malware incursion. Simply put, the aim was to see whether the same observations held: that congestion-aware traffic outnumbered congestion-unaware traffic, in either packets or bytes, on a reasonable view of Internet traffic. Analysis of the these datasets shows that in an Internet backbone link belonging to a Tier 1 ISP, congestion-aware traffic makes up at least 73–82 % of packets, corresponding to 77–84 % of data volume.

[1] This overview might read rather like a polemic against said review—this is merely an attempt to add enough context to make an otherwise dry appendix a tad more entertaining.

A.1 Dataset description

The CAIDA 2018 passive traces are a set of anonymised Pcap files captured over a 1 h period each month from the *equinix-nyc* monitor (observed March–December from 1300–1400 UTC[2]), subdivided into 1 min traces. Source

[2] Depending on daylight savings time, these endpoints fall in UTC-{4,5}↔UTC-3 respectively. For most data points at NYC, this corresponds to 0900–1000 EDT.

and destination IP addresses are anonymised in a prefix-preserving manner. Captured packets have been stripped of their payload data, and remaining headers are accompanied by µs-precise timestamps. Traces are captured over both directions for a monitored Internet backbone link (9953 Mbit/s, Tier 1 ISP) between New York and Sao Paulo: direction A runs from Sao Paulo to New York, while direction B is the reverse of this. The official description contains greater detail (CAIDA, 2018).

A.2 Data processing methodology

Traces are examined at a packet-level granularity, and IPv4 and v6 packets are classified based on their L3/L4 header fields:

TCP Packets with a *protocol* or *next header* value of ***0x06***.

UDP (QUIC) Packets with a *protocol* or *next header* value of ***0x11***, and a source or destination port of 80 or 443.[3]

UDP (Non-QUIC) Packets with a *protocol* or *next header* value of ***0x11*** and any other source and/or destination ports. This includes UDP packets whose ports had been truncated or removed entirely due to IP options.

Other All other packets, including non-IP traffic.

[3] This category is not necessarily definitive, and is based on then-current descriptions from Google on how Chrome was initiating QUIC sessions. These packets aren't *provably* QUIC, but are speculated to be so. CAIDA's payload stripping removes any bytes past the UDP header, making this impossible to conclusively verify.

For each one hour period, I store individual packet counts and payload bytes for each category in ***u64***s. Category counts from sub-traces in the same month are summed together, and once totalled I locally store the counts and proportions of each traffic class (bytes and packets). This is performed for directions A and B separately (*Dir A*, *Dir B*), which are then combined appropriately (*Both*).

Categorisation Congestion-aware packets are defined as those who are either *TCP* or *UDP (QUIC)*. UDP packets (*QUIC* and *Non-QUIC*) are also combined into their own category.

While this analysis ignores other congestion-aware protocols and (cannot be certain on the proportion of QUIC traffic due to the trace data format), this allows us to establish a sensible lower bound on their proportion in the network. Assuming there is no ongoing TCP replay-driven attack in any trace, we can state that the *lower bound* on congestion-aware traffic lies between %(*TCP*) and %(*TCP* + *QUIC*). Similarly, the remainder of these quantities from 1.0 gives us the range of an *upper* bound on congestion-unaware traffic.

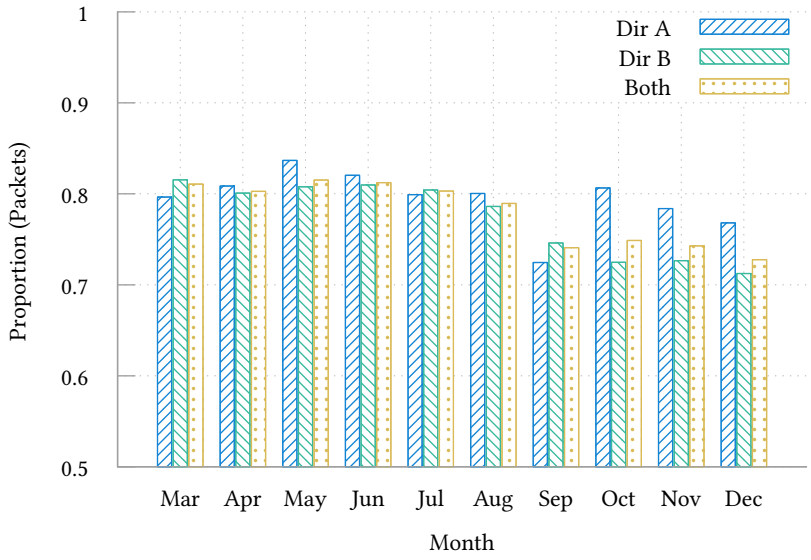
Implementation The above analysis program was implemented in Rust by incrementally streaming, unzipping, and parsing Gzipped packet captures using the *reqwest* library. Individual trace files are mined from the main directory page, which is served as HTML. As network access was the limiting factor in handling this data, these are processed sequentially and in-memory due to the large size of each pcap file (totalling $\mathcal{O}$ (TiB)).

A.3 Results

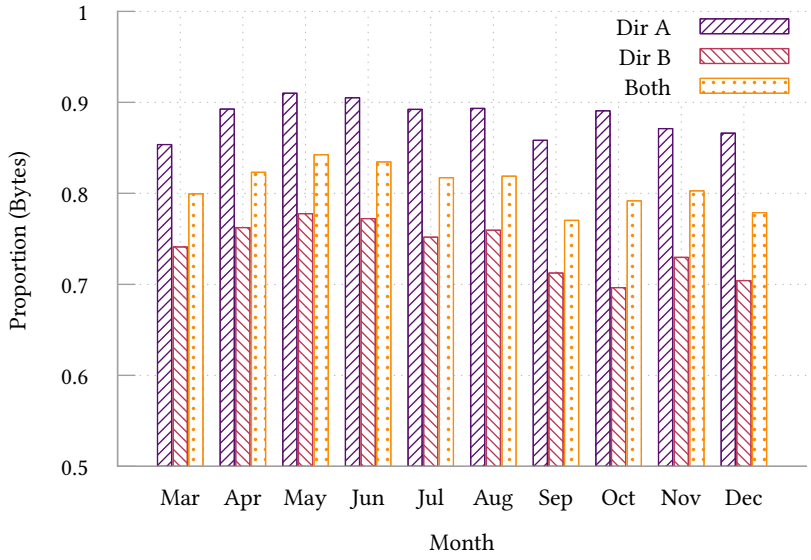
Figure A.1 shows that congestion-aware traffic makes up at least 73–82 % of packets, corresponding to 77–84 % of data volume when considering both transit directions. In *direction A* (Sao Paulo→New York) we can see byte volumes are far greater (85–92 %), while packet counts are roughly symmetric. These trends are effectively replicated for TCP traffic (fig. A.2). The main takeaway is that in both byte volume and proportion of sent packets, TCP still routinely makes up the majority of Internet traffic, and both are higher still for congestion-aware transports.

In the case of UDP traffic including QUIC (fig. A.3), packet counts are again fairly symmetrical—ranging over 15–26 % when considering the aggregate of both directions. Although UDP traffic has a lower packet prevalence in *Dir B*, it consistently has a higher proportion of seen bytes flowing from New York→Sao Paulo.

Supposed QUIC packets do not have much prevalence—particularly in *Dir B* (fig. A.4). We do observe some spikes in activity in *Dir A* ranging over July–September, in 3.6 % of packets and 4.5 % of carried bytes. This falls a little below other estimates of the protocol's prevalence; for instance, Rüth *et al.* (2018) found that it occupied some 2.6–9.1 % of network traffic across another Tier-1 ISP and an IXP. The time of measurement (early-to-mid morning at both endpoints) may be a factor here, as QUIC's main purpose at this time was for video transit rather than the basis of HTTP/3 (Bishop, 2021). The main body type—video—does neatly explain why it occupies a larger share of carried bytes than packets.

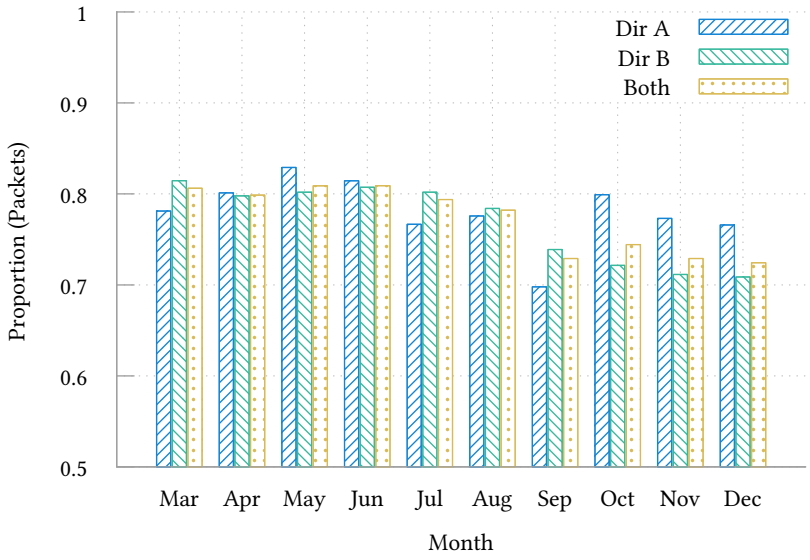


(a) *Congestion-aware packets.*

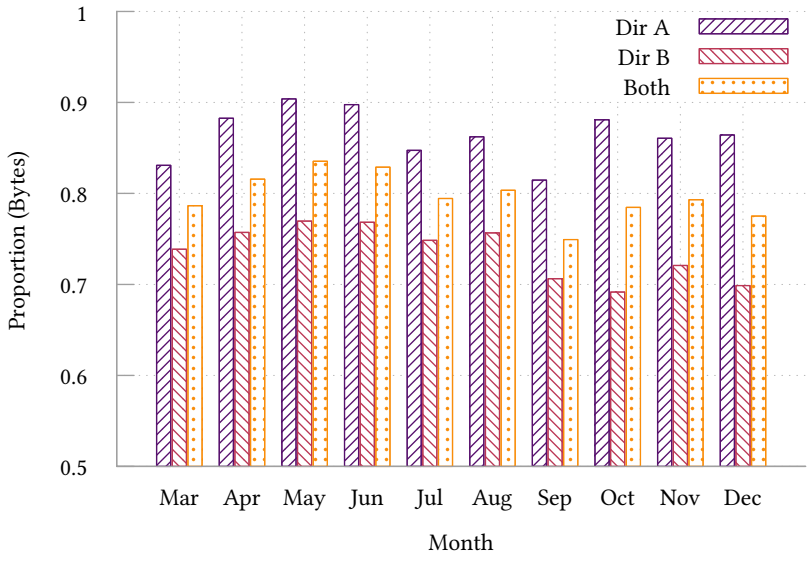


(b) *Congestion-aware byte volume.*

Figure A.1: Proportional counts and byte volume of congestion-aware traffic.

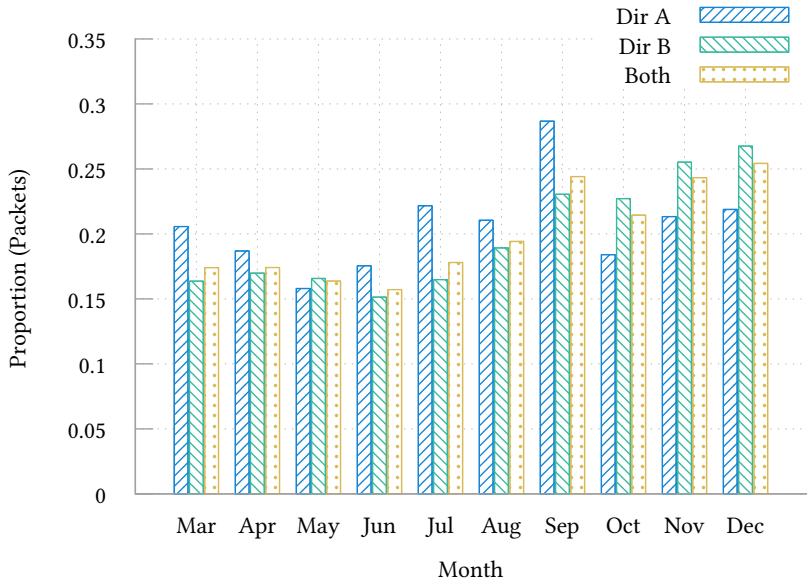


(a) *TCP packets.*

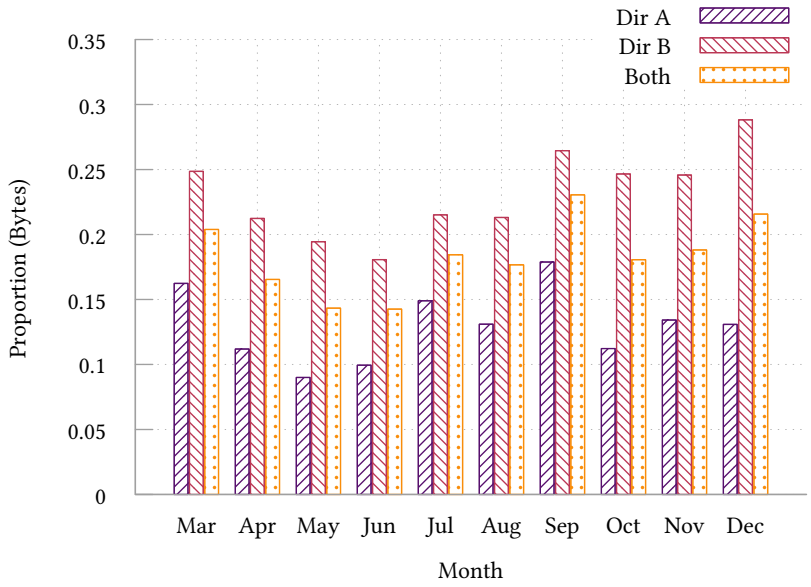


(b) *TCP byte volume.*

Figure A.2: Proportional counts and byte volume of TCP traffic.

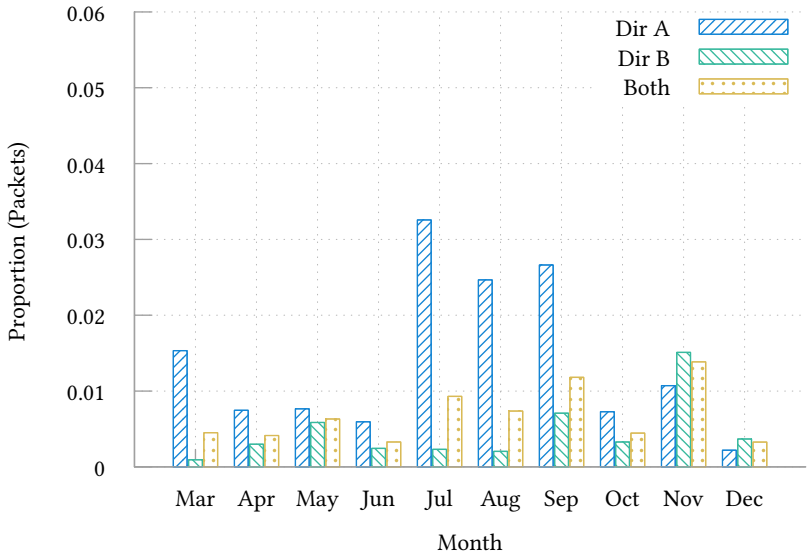


(a) *UDP packets.*

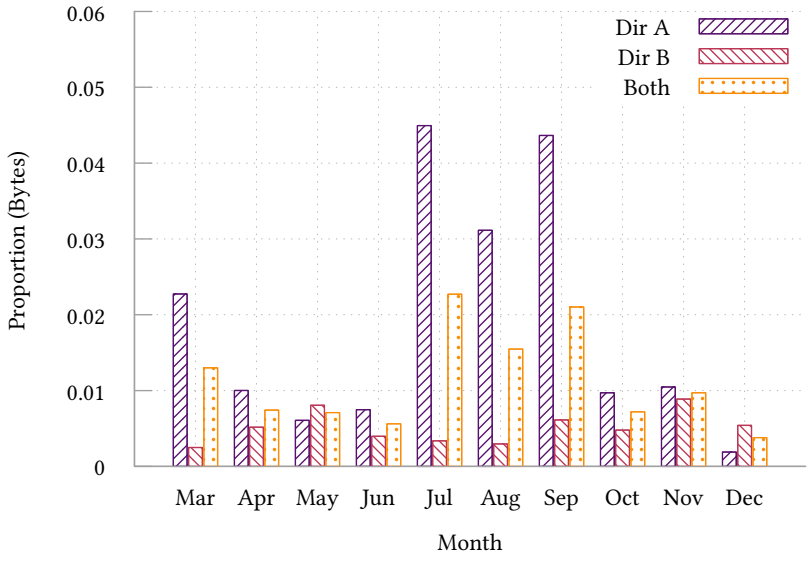


(b) *UDP byte volume.*

Figure A.3: Proportional counts and byte volume of UDP traffic.



(a) QUIC packets.



(b) QUIC byte volume.

Figure A.4: Proportional counts and byte volume of UDP (QUIC) traffic.

Appendix B

Opus VoIP Traffic Capture and Generation

To provide a realistic model of normal congestion-unaware traffic to properly evaluate the work in chapter 4, I passively measured VoIP traffic generated by users of the Discord (Discord, 2022) online chat service to acquire reasonable traces and then generate appropriately similar traffic. Discord allows text, voice and streaming video conversation between users—the latter two allow real-time communication over UDP. I am interested here in voice, which is effectively CBR and congestion-unaware, but is easily accessible by bot users (unlike video).

VoIP application traffic has interesting characteristics. Client→server flows contain an audio stream mixed with in-band and out-of-band control traffic. Audio streams are encoded at a target bitrate (e.g., 96 kbit/s) and divided into individual packets to provide continuous delivery of traffic—these are usually relatively large (~20 ms) to reduce packet transport overhead, but small enough to minimise the impact of packet losses. Due to this real-time requirement, audio packet arrivals and transmissions are then highly periodic. There are other interesting dynamics, aside from the trivial observation that flows won't react substantially to lost packets. In Discord's architecture (and I suspect in the general case nowadays to achieve better and more consistent QoE), *all* packets are sent to a single TURN server (Reddy *et al.*, 2020), which relays them between users to ensure connectivity (as opposed to, say, peer-to-peer session links). This causes inbound RTP packets to fan out to all other participants in a call, leading to a moderate amplification factor.

Sadly, it is insufficient to just encode arbitrary audio at a voice channel's supposed bitrate and then divide that into fixed-size packets. The exact size of each packet depends on the carried content, but tends in the long term towards CBR for speech or music. Silent frames are, of course, the smallest to encode (e.g., 5 B for the Opus codec). Smaller signal-dependent variations aside, since users tend to converse with each other they typically

speak in bursts of various lengths rather than continuously, and can cease sending packets during these times to save bandwidth. Obviously, this leads to burstier traffic than simply taking the expected link occupation over each user's stream. Moreover, the duration of individual voice sessions and number of recipients also play a part in how traffic is fanned out at the TURN server, and to whom.

From the above, we have quite a few factors which affect both client→server and server→client behaviour. At the time I was designing the evaluation for chapter 4, I couldn't find conclusive studies on VoIP traffic which would be at all useful in modelling these sorts of dynamics—and still haven't quite seen any works which fill the same niche. This appendix describes my methodology for capturing and generating somewhat simplified variants of these flows—i.e., discounting control traffic and shared sessions to minimise out-of-band coordination.

B.1 Voice session behaviour

Discord's VoIP sessions target a preset bitrate per voice channel chosen by that server's administrators—8–96 kbit/s on free servers[1], defaulting to 64 kbit/s. Clients connect by requesting session IDs and keys over the main WebSocket API, which are used to open another WebSocket session for voice control traffic. This includes negotiating the cryptographic tag scheme, and receiving an RTP *Synchronisation Source* (SSRC), URL and port for a UDP TURN server. Connection then proceeds to make use of WebRTC (web browser users) or vanilla RTP: in the latter case, explicit NAT hole punching is performed. This WebSocket session is maintained over the call's lifetime to provide information about other users and exchange periodic heartbeat messages.

[1] 128 kbit/s, 256 kbit/s and 384 kbit/s are offered to paying users. This only affects client behaviour: the TURN server is not stringent enough to block or re-encode voice data sent by bots or custom clients.

Audio is encoded using the Opus codec, split into 20 ms RTP (Schulzrinne *et al.*, 2003) packets at the session's target bitrate. In practice, these include RTP extensions to denote (among other functions) hosts' NTP timestamps and per-packet loudness. Clients additionally send random 4 B UDP keepalive values to the TURN server every 5 s, along with *RTP Control Protocol* (RTCP) reports generated at the intervals defined in the specification. RTP and RTCP are multiplexed over the same socket (Perkins & Westerlund, 2010), while payloads are encrypted using XSalsa20-Poly1305 (Bernstein, 2011; Nir & Langley, 2018). The method of doing so is not quite adherent to the Secure RTP specification (Carrara *et al.*, 2004)—encryption and placement of the message authentication code occur using fixed offsets in the parent UDP packet. When users cease speaking, they will send up to 5 silent frames of audio before they stop transmitting packets (resuming on the next significant audio data). Encrypted, multiplexed RTP and RTCP packets from other users are received on the socket used for earlier NAT hole-punching.

B.2 Capture and storage

Traces were captured using two strategies by a voice bot, *Felyne*, which probabilistically played sounds and music from the *Monster Hunter* series of games following simple FSMs. In both cases, consent was given by captured users in a limited set of close-knit servers (general purpose, role-playing, games-focussed) and all traces are fully anonymised to comply with GDPR. The first strategy (*Version I*) was developed and used for chapter 4, while the second (*Version II*) is used in ongoing measurement.

Version I Traces are captured on a per-user or -SSRC basis, to make generation simpler than tracking all call dynamics. During a call, for every packet received from each SSRC I record its RTP timestamp[2], sequence number, and Opus payload size in bytes. Payload sizes do not include RTP extensions or additional bytes required to store message authentication codes or cryptographic nonces. RTCP packets and RTP extensions are discarded.

[2] This is in sample units of the source audio data,

When a user disconnects, or the next packet would cause the timestamp to overflow relative to its first seen value (~1491.3 min for ***u32***s at 960 samples per packet), the session is finalised and stored. To finalise a trace, packet metadata is sorted by its timestamp. I then replace each packet with its payload size, insert 'Missing' markers where expected sequence numbers are not observed, and insert 'Silent' duration markers for valid packet gaps longer than 20 ms.

Version II Traces are captured on a per-call basis, and instead record all RTP and RTCP packet arrivals as a single stream of events timestamped using the system clock. This model is designed to capture the interactions between user voice sessions, rather than simply speaking-silent burst modelling. WebSocket events and timestamps are stored and used to detect user arrivals, departures, and associate multiple SSRCs to individual users in the event of reconnections. For each arrived RTP packet, I capture its arrival time, SSRC, sequence number, RTP timestamp, extension data, and the Opus payload size. Sequence numbers and RTP timestamps are reduced such that every flow's counters begin from 0.

For processing, user IDs and SSRCs are converted into opaque identifiers (i.e., the first seen SSRC is 'user 0', and so on). SSRCs and IDs in all observed WebSocket, RTP, and RTCP packets are replaced with these identifiers. Felyne tracks users who have opted in and out according to per-server configuration (using opt-in 'roles' and explicit global opt-outs): any RTP, RTCP, or speech events from such users are filtered out. Join and disconnect events from these users are not removed, so as to correctly preserve fan-out behaviour of the TURN server for this call A list of all opt-out IDs is stored to make it clear how accurate a trace is (e.g., making it clear whether

packet events are present for 6/8 users over the duration). RTP extensions are kept intact if the type is known to include no personal data (i.e., loudness indicators)—otherwise, extension payloads are zeroed while their types are kept, including those encoded past the one- and two-byte header extensions (Singer *et al.*, 2017). RTCP packets are sanitised such that NTP timestamps begin at zero, and SSRCs are anonymised as above.

B.3 Traffic generation

Currently, RTP traffic generation is only supported using *Version I* traces. I designed a client and server program for this purpose, implementing a simplified form of the session behaviour described above—i.e., without WebSocket control traffic, sender-to-sender coordination of speaking periods, or RTCP packets.

The server program receives UDP traffic on a given port, where it reflects keepalive packets back to their sender and attempts to forward RTP packets to the other recipients in a 'room'. Inbound flow 5-tuples plus SSRCs are assigned to these rooms: each is given a uniformly random capacity from 2–8 participants, and one room-in-progress is held at a time.

Clients randomly draw (without replacement) from the set of all traces, generating RTP packets every 20 ms during speaking phases. Source IP addresses, ports, and SSRCs are randomly generated, and packet bodies are filled with pre-generated random bytes. Sent packets include enough extra bytes on top of the payload to store the Poly1305 message authentication code (16 B). Additionally, hosts punctuate these RTP frames with a 4 B keepalive every 5 s. Due to the lengthy talk and silence bursts introduced by users in tabletop role-playing servers, silent periods are trimmed to a maximum 5 s. Missed packets are handled by calculating an exponentially-weighted moving average over the observed payload sizes. This process continues until the current trace completes *and* the flow has exceeded a user-specified minimum time, at which point a new session is begun. Individual client flows were found to occupy an expected 52.4 kbit/s upstream bandwidth.

Appendix C

Netronome NFP Architectural Details

Netronome Flow Processor (NFP) SmartNICs are many-core *System on a Chip* (SoC) NICs designed for high performance packet processing at up to 40–100 Gbit/s. In concert with other SmartNIC designs, NFP SmartNICs are designed to allow virtually arbitrary packet processing written in the *MicroC* language. This appendix goes into greater detail on these particular SmartNIC devices due to their key role in chapters 5 and 6—primarily because going into meaningful depth concerning device particulars in those chapters would dilute their clarity. Specific quantities, particularly around memory sizes or core counts, refer to the Netronome Agilio LX 1 × 40GbE SmartNIC (containing the NFP-6480 chipset).

C.1 Execution model

Core layout. The Netronome NFP-6480 offers 112 cores, or *Microengines* (MEs), on which arbitrary programs may be run. Cores are clustered into physical groups, termed *islands*, each containing 4 or 12 MEs. There are 7 islands of each size. Each ME runs a single code store and operates at 1.2 GHz, and all 12-ME islands are used by a default P4 pipeline. Generally speaking, MEs are able to communicate with one another and access one another's memory resources or capabilities. As remote accesses, requests, and atomic operations are typically mediated by a shared *Command Push-Pull* (CPP) bus, the cost of doing so typically scales such that cross-island operations are more expensive than island-local. Many islands co-host specific accelerator functions or I/O capabilities, such as the MAC and PCIe bus, a management ARM processor, local memories, and cryptography accelerator units.

Threading. Threads on each ME are known as *contexts*, which are a class of hardware threads. Each ME may choose at compile time to run either 4 or 8 contexts, which then equally divide the register file and LMEM among themselves.[1] As one code store is maintained per ME, all of its child contexts run the same program code though may query the current context number to enable branching behaviour. Contexts are cooperatively scheduled at run time, where context switches are triggered by signalled I/O operations (who must be awaited) or by voluntary yield hints inserted by the programmer. Context switches are effectively zero cost: as the register file is divided among all threads, another thread may instantly progress when the active thread chooses to sleep. Each core offers 15 separate signals which can be independently fired for each context, and a thread may await any or all of a bitset of signals before it resumes execution. These signals may be fired by other MEs, contexts, or by the memory units in response to a completed I/O operation.

[1] In some cases, per-context resource use may be too great to allow all threads to operate.

Programming. NFP devices support a proprietary assembler language, and a variant of the C programming language termed *MicroC*. This constitutes C with some additions, including an explicit memory model tailored towards this device, signalling and signal datatypes, and agressive inlining capabilities. P4 programs may be compiled to target the NFP, at which point they are compiled into a selection of MicroC programs installed across most available islands. Accordingly, P4 `extern`s resolve to MicroC functions which are arbitrarily defined and included by the programmer.

C.2 Memory

Table C.1 outlines the primary memory regions available, organised in terms of memory cost (where all registers are equal). As above, these register files are split among all contexts at compile time. Xfer registers are not usable outside of their purpose as holding space for the source and destination for I/O operations (and are visible to other MEs and memory units). Next-neighbour registers allow very fast writes between adjacent MEs in an island. These allow MEs to communicate in one of two orders: *chain* (0→1→2→3), and *alternate* (0→2→1→3). Note that this communication is unidirectional and does not form a cycle. Additionally, this functionality may be disabled on a per-ME basis to provide additional register space.

Memories outside of EMEM are small in line with the typical expectations surrounding resource-limited environments like PDP hardware. While almost all per-island or shared memories may be accessed from remote islands, cross-island accesses are more expensive and are typically avoided. Langlet (2019, p. 30) relates his own measurements of these costs, save for EMEM Cache which is allocated and accessed solely—to the best of my knowledge—by the compiler. My understanding is that EMEM Cache is primarily occu-

Table C.1: NFP memory hierarchy, locations, and sizes.

Memory Region	Location	Remote Access	Size
Register (GPR)	Per-ME	✗	2 KiB
Register (Xfer)	Per-ME	✓	1 KiB In, 1 KiB Out
Register (NN)	Per-ME	✗	512 B
LMEM	Per-ME	✗	4 KiB
CLS	Per-Island	✓	64 KiB
CTM	Per-Island	✓	256 KiB
IMEM	i28, i29	✓	4 MiB
EMEM Cache	i24, i25, i26	✓	3 MiB
EMEM	i24, i25	✓	4 GiB, 3.5 GiB

pied by *Content-Addressable Memory* (CAM)-accelerated lookups offered via the provided hash table primitives.

Appendix D

OPaL Control Protocol

OPaL's control protocol is carried within UDP packets. Its presence is signalled to the P4 control plane by setting the DSCP field of the IP header to `0b000011`, though in practice this choice is fairly arbitrary and only serves to allow for easy detection and filtering in the network without impacting valid user choices of more common fields such as UDP port number.

Firstly, we choose a fixed-point representation type at compile time, setting `type Tile` $\in$ {`i8`, `i16`, `i32`}. These and any other numeric types are stored in big-endian format. To minimise packet size, it is assumed that the sender is aware of the datatype employed by the target OPaL agent. Any fields marked with a $\star$ are of type `Tile` and scale according to `sizeof(Tile)` (affecting the offset of all subsequent fields). Any fields marked with a $\diamond$ are of type `Tile` and are zero-padded to 4 B. Packet diagrams display these layouts assuming that quantised numbers are 4 B wide.

Configuration. Configuration of OPaL is managed using two classes of packet: *setup* (fig. D.1) and *tilings* (fig. D.3). Setup packets contain a mixture of operational and policy structure parameters. While most of these fields are self-explanatory, they behave as follows:

F Forces RL update logic to occur if set to 1, even if a valid historic state and reward pair cannot be found.

N Disables writeout of inferred state-action pairs over the OUT ring if set to 1.

O Enables online learning if set to 1.

shift_amt The number of fractional bits in each fixed-point number.

worker_limit A software limit on active worker threads. A setting of 0 disables this limit.

n_dims The total number of dimensions expected in state vectors.

tiles_per_dim The number of tiles which every dimension is subdivided into.

tilings_per_set The number of tilings to offset and stride across each list of dimensions.

n_actions The number of output actions to select between.

$\epsilon^\star$ The current chance of selecting a randomised action (i.e., ϵ-greedy action selection).

$\alpha^\star$ The learning rate as in eq. (3.6).

$\gamma^\star$ The discount factor as in eq. (3.6).

$\epsilon^\star_{\textit{decay amount}}$ The amount by which ϵ should be decreased every time it is annealed.

$\epsilon_{\textit{decay frequency}}$ The number of actions to wait before decreasing ϵ.

state_key The selection method for retrieving historic state-action tuples mapped to an input state (i.e., execution trajectories).

reward_key The selection method for retrieving the reward value mapped to an input state.

maxes* The maximum value allowed in a state vector for each input dimension.

mins* The minimum value allowed in a state vector for each input dimension.

Of these, state/reward key lookups (fig. D.2) admit 3 types. Keys may be retrieved as a single shared value, ignoring the `location` field (type 0). Alternately, they may admit a field of the input state as the key (type 1) retrieving, e.g., `rewards[hash(input[location])]`. They may directly access the storage map (type 2) retrieving, e.g., `rewards[input[location]]`. Finally, *reward* values may be accessed as a field in the input vector (type 4), where `location` is the index in the state vector to select—this obviously cannot extend to state lookup. These correspond to *Shared*, *Field*, *Raw Field*, and *Value* respectively as covered by section 5.1.1.

Tiling packets are composed of a list of individual tiling instances (fig. D.4), parsed until the end of the UDP datagram. Each tiling instance contains a length `dim_list_len`, a `location` $\in [0, 2]$ (CLS, CTM or IMEM according to section 5.2.1), and a list of `dim_list_len` state indices to be used as tiling dimensions. These instances must be present in location-sorted order, smallest to largest. Additionally, dimension lists' size must not exceed the limit for their parent memory region (1, 2, and 4 dims respectively).

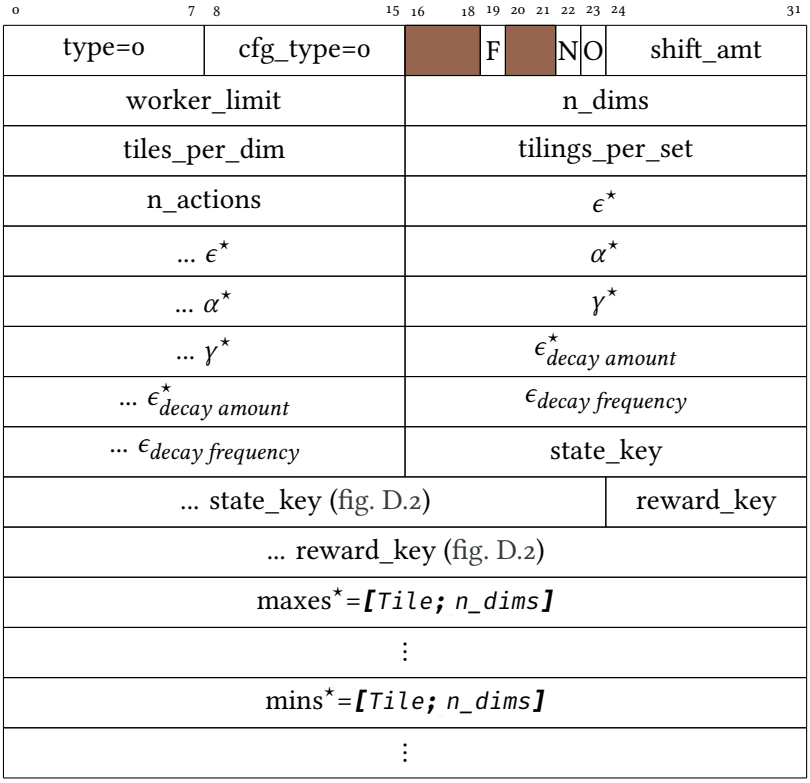


Figure D.1: OPaL configuration (setup) packet.

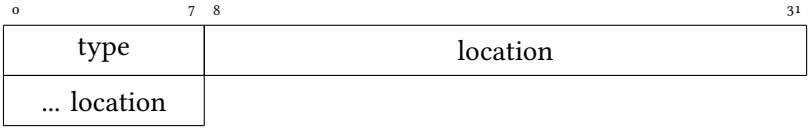


Figure D.2: OPaL lookup key source layout.

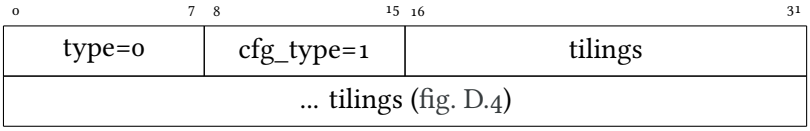


Figure D.3: OPaL configuration (tiling) packet.

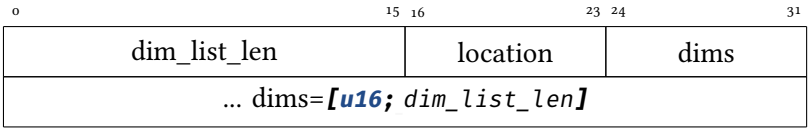


Figure D.4: OPaL tiling instance layout.

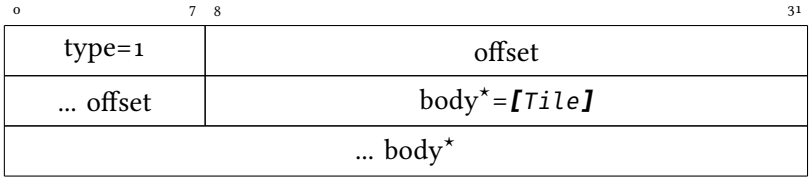


Figure D.5: OPaL policy insertion header.

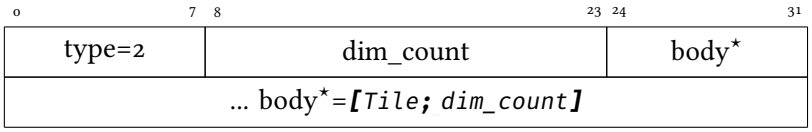


Figure D.6: OPaL state vector packet.

Policy Insertion. *Insert* packets (fig. D.5) contain an `offset`—the index of the first policy value contained in this packet—and are then filled for the remainder of the datagram with tile values (`body`). These packets are free to straddle memory region boundaries, be unaligned with respect to actions in a tiling, and arrive in any order.

State Vectors. *State* packets (fig. D.6) are used to pass in state from the network to OPaL, and simply contain a list of `Tile`s of size `dim_count`.

Reward Measurements. *Reward* packets contain a reward `value` of type `Tile` padded to 4 B. If reward lookups rely on state vector fields to match a trace (*Field* or *Raw Field*) then `key` is used to store this value in the correct location.

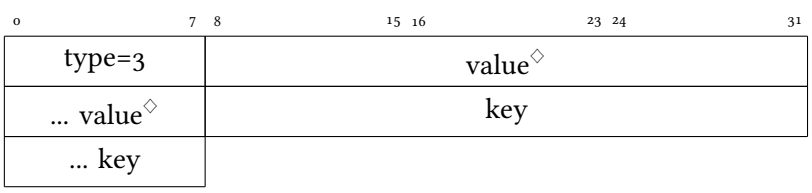


Figure D.7: OPaL reward header.

References

Abadi, M., Chu, A., Goodfellow, I. J., McMahan, H. B., Mironov, I., Talwar, K., & Zhang, L. (2016). Deep Learning with Differential Privacy. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers & S. Halevi (Eds.), *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016* (pp. 308–318). ACM. https://doi.org/10.1145/2976749.2978318 (see p. 106).

Abbasloo, S., Yen, C., & Chao, H. J. (2020). Classic Meets Modern: a Pragmatic Learning-Based Congestion Control for the Internet. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 632–647). ACM. https://doi.org/10.1145/3387514.3405892 (see p. 66).

Aceto, G., Ciuonzo, D., Montieri, A., & Pescapè, A. (2019). Mobile Encrypted Traffic Classification Using Deep Learning: Experimental Evaluation, Lessons Learned, and Challenges. *IEEE Trans. Network and Service Management*, *16*(2), 445–458. https://doi.org/10.1109/TNSM.2019.2899085 (see p. 64).

Agere Systems. (2001). *PayloadPlus™ Fast Pattern Processor*. Retrieved February 7, 2022, from https://web.archive.org/web/20220207202633/http://static6.arrow.com/aropdfconversion/42adcb01fbfd55f6b6a512b93dba2ea92f065a76/fpp_product_brief.pdf (see p. 16).

Aghakhani, H., Meng, D., Wang, Y., Kruegel, C., & Vigna, G. (2020). Bullseye Polytope: A Scalable Clean-Label Poisoning Attack with Improved Transferability. *CoRR*, *abs/2005.00191*. https://arxiv.org/abs/2005.00191 (see p. 103).

Aitken, P., Claise, B., & Trammell, B. (2013). Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of Flow Information. https://doi.org/10.17487/RFC7011 (see pp. 49, 191).

Alexander, D. S., Arbaugh, W., Hicks, M., Kakkar, P., Keromytis, A. D., Moore, J. T., Gunter, C. A., Nettles, S., & Smith, J. M. (1998). The SwitchWare active network architecture. *IEEE Netw.*, *12*(3), 29–36. https://doi.org/10.1109/65.690959 (see p. 15).

Alexander, D. S., Arbaugh, W., Keromytis, A. D., & Smith, J. M. (1998). A secure active network environment architecture: realization in SwitchWare. *IEEE Netw.*, *12*(3), 37–45. https://doi.org/10.1109/65.690960 (see p. 15).

Al-Fares, M., Loukissas, A., & Vahdat, A. (2008). A scalable, commodity data center network architecture. In V. Bahl, D. Wetherall, S. Savage & I. Stoica (Eds.), *Proceedings of the ACM SIGCOMM 2008 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Seattle, WA, USA, August 17-22, 2008* (pp. 63–74). ACM. https://doi.org/10.1145/1402958.1402967 (see p. 136).

Alipourfard, O., Liu, H. H., Chen, J., Venkataraman, S., Yu, M., & Zhang, M. (2017). CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In A. Akella & J. Howell (Eds.), *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017* (pp. 469–482). USENIX Association. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/alipourfard (see p. 64).

Amir, G., Schapira, M., & Katz, G. (2021). Towards Scalable Verification of Deep Reinforcement Learning [To appear.]. *2021 Formal Methods in Computer Aided Design, FMCAD 2021, Yale University, MA, USA, October 20–22, 2021*. https://a95dd233-6d9a-4a97-bf9f-a1133293b480.filesusr.com/ugd/3b1e1e_56e99c5c72fe4cbda05c5475bfb08de3.pdf (see p. 97).

Antonakakis, M., April, T., Bailey, M., Bernhard, M., Bursztein, E., Cochran, J., Durumeric, Z., Halderman, J. A., Invernizzi, L., Kallitsis, M., Kumar, D., Lever, C., Ma, Z., Mason, J., Menscher, D., Seaman, C., Sullivan, N., Thomas, K., & Zhou, Y. (2017). Understanding the Mirai Botnet. In E. Kirda & T. Ristenpart (Eds.), *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017.* (pp. 1093–1110). USENIX Association. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis (see pp. 111, 114).

Arashloo, M. T., Lavrov, A., Ghobadi, M., Rexford, J., Walker, D., & Wentzlaff, D. (2020). Enabling Programmable Transport Protocols in High-Speed NICs. In R. Bhagwan & G. Porter (Eds.), *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (pp. 93–109). USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/arashloo (see p. 54).

Arun, V., & Balakrishnan, H. (2018). Copa: Practical Delay-Based Congestion Control for the Internet. In S. Banerjee & S. Seshan (Eds.), *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (pp. 329–342). USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/arun (see pp. 53, 65).

Athalye, A., Engstrom, L., Ilyas, A., & Kwok, K. (2017). Synthesizing Robust Adversarial Examples. *CoRR, abs/1707.07397*. http://arxiv.org/abs/1707.07397 (see p. 99).

Auld, T., Moore, A. W., & Gull, S. F. (2007). Bayesian Neural Networks for Internet Traffic Classification. *IEEE Trans. Neural Networks*, *18*(1), 223–239. https://doi.org/10.1109/TNN.2006.883010 (see p. 64).

Axelsson, S. (1999). The Base-Rate Fallacy and Its Implications for the Difficulty of Intrusion Detection. In J. Motiwalla & G. Tsudik (Eds.), *CCS '99, Proceedings of the 6th ACM Conference on Computer and Communications Security, Singapore, November 1-4, 1999*. (pp. 1–7). ACM. https://doi.org/10.1145/319709.319710 (see p. 67).

Azar, Y., Cohen, E., Fiat, A., Kaplan, H., & Räcke, H. (2003). Optimal oblivious routing in polynomial time. In L. L. Larmore & M. X. Goemans (Eds.), *Proceedings of the 35th Annual ACM Symposium on Theory of Computing, June 9-11, 2003, San Diego, CA, USA* (pp. 383–388). ACM. https://doi.org/10.1145/780542.780599 (see p. 61).

Bagdasaryan, E., Veit, A., Hua, Y., Estrin, D., & Shmatikov, V. (2020). How To Backdoor Federated Learning. In S. Chiappa & R. Calandra (Eds.), *The 23rd International Conference on Artificial Intelligence and Statistics, AISTATS 2020, 26-28 August 2020, Online [Palermo, Sicily, Italy]* (pp. 2938–2948, Vol. 108). PMLR. http://proceedings.mlr.press/v108/bagdasaryan20a.html (see p. 103).

Bahnasy, M., Li, F., Xiao, S., & Cheng, X. (2020). DeepBGP: A Machine Learning Approach for BGP Configuration Synthesis. In B. Arzani & X. Jin (Eds.), *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM, Virtual Event, USA, August 14, 2020* (pp. 48–55). ACM. https://doi.org/10.1145/3405671.3405816 (see p. 63).

Bajaj, S., Breslau, L., Estrin, D., Fall, K., Floyd, S., Haldar, P., Handley, M., Helmy, A., Heidemann, J., Huang, P., Kumar, S., McCanne, S., Yu, H., & et al. (1999, March 4). *Improving Simulation for Network Research* (tech. rep.). Univeristy of Southern California. Retrieved May 22, 2022, from https://zappala.byu.edu/static/pubs/usc-cs-tr-99-702.pdf (see p. 19).

Baker, F., Black, D. L., Nichols, K., & Blake, S. L. (1998). Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. https://doi.org/10.17487/RFC2474 (see pp. 56, 169).

Barbette, T., Soldani, C., & Mathy, L. (2015). Fast Userspace Packet Processing. In G. J. Brebner, A. Bachmutsky & C. R. Das (Eds.), *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7-8, 2015* (pp. 5–16). IEEE Computer Society. https://doi.org/10.1109/ANCS.2015.7110116 (see p. 24).

Barbette, T., Tang, C., Yao, H., Kostic, D., Jr., G. Q. M., Papadimitratos, P., & Chiesa, M. (2020). A High-Speed Load-Balancer Design with Guaranteed Per-Connection-Consistency. In R. Bhagwan &

G. Porter (Eds.), *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (pp. 667–683). USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/barbette (see p. 51).

Barefoot. (2017). *The World's Fastest & Most Programmable Networks*. Retrieved February 4, 2022, from https://web.archive.org/web/20200528151742/https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/ (see p. 27).

Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., & Warfield, A. (2003). Xen and the art of virtualization. In M. L. Scott & L. L. Peterson (Eds.), *Proceedings of the 19th ACM Symposium on Operating Systems Principles 2003, SOSP 2003, Bolton Landing, NY, USA, October 19-22, 2003* (pp. 164–177). ACM. https://doi.org/10.1145/945445.945462 (see p. 24).

Barreno, M., Nelson, B., Sears, R., Joseph, A. D., & Tygar, J. D. (2006). Can machine learning be secure? In F. Lin, D. Lee, B. P. Lin, S. Shieh & S. Jajodia (Eds.), *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security, ASIACCS 2006, Taipei, Taiwan, March 21-24, 2006* (pp. 16–25). ACM. https://doi.org/10.1145/1128817.1128824 (see p. 98).

Barroso, L. A., Marty, M., Patterson, D. A., & Ranganathan, P. (2017). Attack of the killer microseconds. *Commun. ACM*, *60*(4), 48–54. https://doi.org/10.1145/3015146 (see p. 39).

Barth-Maron, G., Hoffman, M. W., Budden, D., Dabney, W., Horgan, D., TB, D., Muldal, A., Heess, N., & Lillicrap, T. P. (2018). Distributed Distributional Deterministic Policy Gradients. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. https://openreview.net/forum?id=SyZipzbCb (see p. 89).

Bartulovic, M., Jiang, J., Balakrishnan, S., Sekar, V., & Sinopoli, B. (2017). Biases in Data-Driven Networking, and What to Do About Them. In S. Banerjee, B. Karp & M. Walfish (Eds.), *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017* (pp. 192–198). ACM. https://doi.org/10.1145/3152434.3152448 (see p. 95).

Bas, A. (2016). *BEHAVIORAL MODEL (bmv2): The reference P4 software switch*. Retrieved March 4, 2022, from https://github.com/p4lang/behavioral-model (see p. 28).

Basat, R. B., Ramanathan, S., Li, Y., Antichi, G., Yu, M., & Mitzenmacher, M. (2020). PINT: Probabilistic In-band Network Telemetry. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 662–680). ACM. https://doi.org/10.1145/3387514.3405894 (see pp. 29, 50).

Bauer, S., Jaeger, B., Helfert, F., Barias, P., & Carle, G. (2021). On the evolution of internet flow characteristics. *ANRW '21: Applied Networking Research Workshop, Virtual Event, USA, July 24-30, 2021*, 29–35. https://doi.org/10.1145/3472305.3472321 (see pp. 115, 212).

Bellman, R. E. (1957). A Markovian Decision Process. *Journal of Mathematics and Mechanics*, *6*(5), 679–684. http://www.jstor.org/stable/24900506 (see p. 85).

Bernaille, L., Teixeira, R., Akodkenou, I., Soule, A., & Salamatian, K. (2006). Traffic classification on the fly. *Comput. Commun. Rev.*, *36*(2), 23–26. https://doi.org/10.1145/1129582.1129589 (see p. 191).

Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., Józefowicz, R., Gray, S., Olsson, C., Pachocki, J., Petrov, M., de Oliveira Pinto, H. P., Raiman, J., Salimans, T., Schlatter, J., … Zhang, S. (2019). Dota 2 with Large Scale Deep Reinforcement Learning. *CoRR*, *abs/1912.06680*. http://arxiv.org/abs/1912.06680 (see p. 3).

Bernstein, D. J. (2011, February 4). Extending the Salsa20 Nonce. In G. Leander & S. S. Thomsen (Eds.), *Proceedings of the Symmetric Key Encryption Workshop (SKEW) 2011, February 16–17, 2011*. http://cr.yp.to/snuffle/xsalsa-20110204.pdf (see p. 224).

Bhuyan, M. H., Bhattacharyya, D. K., & Kalita, J. K. (2014). Network Anomaly Detection: Methods, Systems and Tools. *IEEE Communications Surveys and Tutorials*, *16*(1), 303–336. https://doi.org/10.1109/SURV.2013.052213.00046 (see pp. 67, 109).

Bishop, M. (2021). *Hypertext Transfer Protocol Version 3 (HTTP/3)* (Internet-Draft No. draft-ietf-quic-http-34) (Work in Progress). Internet Engineering Task Force. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34 (see p. 217).

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., & Walker, D. (2014). P4: programming protocol-independent packet processors. *Computer Communication Review*, *44*(3), 87–95. https://doi.org/10.1145/2656877.2656890 (see p. 27).

Bosshart, P., Gibb, G., Kim, H., Varghese, G., McKeown, N., Izzard, M., Mujica, F. A., & Horowitz, M. (2013). Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN. In D. M. Chiu, J. Wang, P. Barford & S. Seshan (Eds.), *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013* (pp. 99–110). ACM. https://doi.org/10.1145/2486001.2486011 (see pp. 2, 27).

Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization Methods for Large-Scale Machine Learning. *SIAM Rev.*, *60*(2), 223–311. https://doi.org/10.1137/16M1080173 (see p. 84).

Bourtoule, L., Chandrasekaran, V., Choquette-Choo, C. A., Jia, H., Travers, A., Zhang, B., Lie, D., & Papernot, N. (2021). Machine Unlearning. *42nd IEEE Symposium on Security and Privacy, SP 2021,*

San Francisco, CA, USA, 24-27 May 2021, 141–159. https://doi.org/10.1109/SP40001.2021.00019 (see p. 106).

Bradner, S., & McQuaid, J. (1999). Benchmarking Methodology for Network Interconnect Devices. https://doi.org/10.17487/RFC2544 (see p. 174).

Braga, R., de Souza Mota, E., & Passito, A. (2010). Lightweight DDoS flooding attack detection using NOX/OpenFlow. In *The 35th Annual IEEE Conference on Local Computer Networks, LCN 2010, 10-14 October 2010, Denver, Colorado, USA, Proceedings* (pp. 408–415). IEEE Computer Society. https://doi.org/10.1109/LCN.2010.5735752 (see pp. 68, 116).

Breiman, L. (2001). Random Forests. *Mach. Learn.*, *45*(1), 5–32. https://doi.org/10.1023/A:1010933404324 (see p. 68).

Bremler-Barr, A., Harchol, Y., & Hay, D. (2016). OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In M. P. Barcellos, J. Crowcroft, A. Vahdat & S. Katti (Eds.), *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (pp. 511–524). ACM. https://doi.org/10.1145/2934872.2934875 (see p. 24).

Brunella, M. S., Belocchi, G., Bonola, M., Pontarelli, S., Siracusano, G., Bianchi, G., Cammarano, A., Palumbo, A., Petrucci, L., & Bifulco, R. (2020). hXDP: Efficient Software Packet Processing on FPGA NICs. *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 973–990. https://www.usenix.org/conference/osdi20/presentation/brunella (see p. 47).

Caesar, M., Caldwell, D. F., Feamster, N., Rexford, J., Shaikh, A., & van der Merwe, J. E. (2005). Design and Implementation of a Routing Control Platform. In A. Vahdat & D. Wetherall (Eds.), *2nd Symposium on Networked Systems Design and Implementation (NSDI 2005), May 2-4, 2005, Boston, Massachusetts, USA, Proceedings*. USENIX. http://www.usenix.org/events/nsdi05/tech/caesar.html (see p. 19).

Cai, Q., Chaudhary, S., Vuppalapati, M., Hwang, J., & Agarwal, R. (2021). Understanding host network stack overheads. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021* (pp. 65–77). ACM. https://doi.org/10.1145/3452296.3472888 (see pp. 40, 41).

CAIDA. (2018). *The CAIDA UCSD Anonymized Internet Traces – 2018*. Retrieved May 11, 2019, from http://www.caida.org/data/passive/passive_dataset.xml (see pp. 115, 124, 215, 216).

Calvert, K. L. (2006). Reflections on network architecture: an active networking perspective. *Comput. Commun. Rev.*, *36*(2), 27–30. https://doi.org/10.1145/1129582.1129590 (see p. 14).

Campbell, A. T., Katzela, I., Miki, K., & Vicente, J. B. (1999). Open signaling for ATM, internet and mobile networks (OPENSIG'98). *Comput.*

Commun. Rev., *29*(1), 97–108. https://doi.org/10.1145/505754.505762 (see p. 18).

Cao, X., & Gong, N. Z. (2017). Mitigating Evasion Attacks to Deep Neural Networks via Region-based Classification. In *Proceedings of the 33rd Annual Computer Security Applications Conference, Orlando, FL, USA, December 4-8, 2017* (pp. 278–287). ACM. https://doi.org/10.1145/3134600.3134606 (see p. 101).

Cardwell, N., Cheng, Y., Gunn, C. S., Yeganeh, S. H., & Jacobson, V. (2016). BBR: Congestion-Based Congestion Control. *ACM Queue*, *14*(5), 20–53. https://doi.org/10.1145/3012426.3022184 (see pp. 2, 53, 115, 192, 199).

Carlini, N. (2021). Poisoning the Unlabeled Dataset of Semi-Supervised Learning. In M. Bailey & R. Greenstadt (Eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (pp. 1577–1592). USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/carlini-poisoning (see p. 102).

Carlini, N., Athalye, A., Papernot, N., Brendel, W., Rauber, J., Tsipras, D., Goodfellow, I. J., Madry, A., & Kurakin, A. (2019). On Evaluating Adversarial Robustness. *CoRR*, *abs/1902.06705*. http://arxiv.org/abs/1902.06705 (see p. 100).

Carlini, N., Tramèr, F., Wallace, E., Jagielski, M., Herbert-Voss, A., Lee, K., Roberts, A., Brown, T. B., Song, D., Erlingsson, Ú., Oprea, A., & Raffel, C. (2020). Extracting Training Data from Large Language Models. *CoRR*, *abs/2012.07805*. https://arxiv.org/abs/2012.07805 (see p. 105).

Carlini, N., & Wagner, D. A. (2017). Towards Evaluating the Robustness of Neural Networks. In *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017* (pp. 39–57). IEEE Computer Society. https://doi.org/10.1109/SP.2017.49 (see pp. 99, 118).

Carrara, E., Norrman, K., McGrew, D., Naslund, M., & Baugher, M. (2004). The Secure Real-time Transport Protocol (SRTP). https://doi.org/10.17487/RFC3711 (see p. 224).

Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., & Shenker, S. (2007). Ethane: taking control of the enterprise. In J. Murai & K. Cho (Eds.), *Proceedings of the ACM SIGCOMM 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Kyoto, Japan, August 27-31, 2007* (pp. 1–12). ACM. https://doi.org/10.1145/1282380.1282382 (see p. 19).

Cavium. (2017). *XPliant Ethernet Switch Product Family*. Retrieved February 24, 2022, from https://web.archive.org/web/20170713110608/http://cavium.com/XPliant-Ethernet-Switch-Product-Family.html (see p. 27).

Celis, P., Larson, P., & Munro, J. I. (1985). Robin Hood Hashing (Preliminary Report). *26th Annual Symposium on Foundations of Computer*

Science, Portland, Oregon, USA, 21-23 October 1985, 281–288. https://doi.org/10.1109/SFCS.1985.48 (see p. 194).

Chandrasekaran, V., Chaudhuri, K., Giacomelli, I., Jha, S., & Yan, S. (2020). Exploring Connections Between Active Learning and Model Extraction. In S. Capkun & F. Roesner (Eds.), *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (pp. 1309–1326). USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/chandrasekaran (see p. 106).

Chen, L., Lingys, J., Chen, K., & Liu, F. (2018). AuTO: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In S. Gorinsky & J. Tapolcai (Eds.), *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (pp. 191–205). ACM. https://doi.org/10.1145/3230543.3230551 (see pp. 62, 125, 141).

Chen, X., Feibish, S. L., Koral, Y., Rexford, J., & Rottenstreich, O. (2018). Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks, SelfDN@SIGCOMM 2018, Budapest, Hungary, August 24, 2018* (pp. 22–28). ACM. https://doi.org/10.1145/3229584.3229586 (see p. 49).

Chen, X., Liu, C., Li, B., Lu, K., & Song, D. (2017). Targeted Backdoor Attacks on Deep Learning Systems Using Data Poisoning. *CoRR, abs/1712.05526*. http://arxiv.org/abs/1712.05526 (see p. 103).

Chiosi, M., Clarke, D., Willis, P., Reid, A., Feger, J., Bugenhagen, M., Khan, W., Fargano, M., Cui, C., Deng, H., Benitez, J., Michel, U., Damker, H., Ogaki, K., Matsuzaki, T., Fukui, M., Shimano, K., Delisle, D., Loudier, Q., … Sen, P. (2012). *Network Functions Virtualisation: An Introduction, Benefits, Enablers, Challenges & Call for Action* (tech. rep.). Retrieved February 24, 2022, from https://portal.etsi.org/nfv/nfv_white_paper.pdf (see p. 23).

Cho, K., van Merrienboer, B., Gülçehre, Ç., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In A. Moschitti, B. Pang & W. Daelemans (Eds.), *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL* (pp. 1724–1734). ACL. https://doi.org/10.3115/v1/d14-1179 (see p. 82).

Chrabaszcz, P., Loshchilov, I., & Hutter, F. (2018). Back to Basics: Benchmarking Canonical Evolution Strategies for Playing Atari. *CoRR, abs/1802.08842*. http://arxiv.org/abs/1802.08842 (see p. 88).

Chun, B. N., Culler, D. E., Roscoe, T., Bavier, A. C., Peterson, L. L., Wawrzoniak, M., & Bowman, M. (2003). PlanetLab: an overlay testbed for broad-coverage services. *Comput. Commun. Rev.*, *33*(3), 3–12. https://doi.org/10.1145/956993.956995 (see p. 14).

Cisco. (2014). *Cisco Event Response: Network Time Protocol Amplification Distributed Denial of Service Attacks*. Retrieved September 23, 2019, from https://www.cisco.com/c/en/us/about/security-center/event-response/network-time-protocol-amplification-ddos.html (see p. 135).

Claise, B. (2004). Cisco Systems NetFlow Services Export Version 9. https://doi.org/10.17487/RFC3954 (see pp. 49, 191).

Clark, D. D., Partridge, C., Ramming, J. C., & Wroclawski, J. (2003). A knowledge plane for the internet. In A. Feldmann, M. Zitterbart, J. Crowcroft & D. Wetherall (Eds.), *Proceedings of the ACM SIGCOMM 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, August 25-29, 2003, Karlsruhe, Germany* (pp. 3–10). ACM. https://doi.org/10.1145/863955.863957 (see p. 60).

Cohen, T. (2015, September 30). *Israel's Mellanox to buy EZchip for $811 million*. Retrieved March 4, 2022, from https://www.reuters.com/article/us-ezchip-sem-m-a-mellanox-idUSKCN0RU1DN20150930 (see p. 25).

Collins, M. P., Shimeall, T. J., Faber, S., Janies, J., Weaver, R., Shon, M. D., & Kadane, J. B. (2007). Using uncleanliness to predict future botnet addresses. In C. Dovrolis & M. Roughan (Eds.), *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference, IMC 2007, San Diego, California, USA, October 24-26, 2007* (pp. 93–104). ACM. https://doi.org/10.1145/1298306.1298319 (see pp. 113, 131).

Copty, F., Danos, M., Edelstein, O., Eisner, C., Murik, D., & Zeltser, B. (2018). Accurate Malware Detection by Extreme Abstraction. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018* (pp. 101–111). ACM. https://doi.org/10.1145/3274694.3274700 (see pp. 68, 99).

Corbet, J. (2018, April 9). *Accelerating networking with AF_XDP*. Retrieved March 17, 2022, from https://lwn.net/Articles/750845/ (see p. 43).

Courbariaux, M., Bengio, Y., & David, J. (2015). Low precision arithmetic for deep learning. In Y. Bengio & Y. LeCun (Eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. http://arxiv.org/abs/1412.7024 (see p. 93).

Cziva, R., Anagnostopoulos, C., & Pezaros, D. P. (2018). Dynamic, Latency-Optimal vNF Placement at the Network Edge. *2018 IEEE Conference on Computer Communications, INFOCOM 2018, Honolulu, HI, USA, April 16-19, 2018*, 693–701. https://doi.org/10.1109/INFOCOM.2018.8486021 (see p. 24).

Cziva, R., Jouët, S., White, K. J. S., & Pezaros, D. P. (2015). Container-based network function virtualization for software-defined networks. *2015 IEEE Symposium on Computers and Communication, ISCC 2015, Larnaca, Cyprus, July 6-9, 2015*, 415–420. https://doi.org/10.1109/ISCC.2015.7405550 (see p. 24).

Cziva, R., & Pezaros, D. P. (2017). Container Network Functions: Bringing NFV to the Network Edge. *IEEE Commun. Mag.*, *55*(6), 24–31. https://doi.org/10.1109/MCOM.2017.1601039 (see pp. 24, 183).

Czyz, J., Kallitsis, M., Gharaibeh, M., Papadopoulos, C., Bailey, M., & Karir, M. (2014). Taming the 800 Pound Gorilla: The Rise and Decline of NTP DDoS Attacks. In C. Williamson, A. Akella & N. Taft (Eds.), *Proceedings of the 2014 Internet Measurement Conference, IMC 2014, Vancouver, BC, Canada, November 5-7, 2014* (pp. 435–448). ACM. https://doi.org/10.1145/2663716.2663717 (see pp. 111, 113).

D'Ambrosia, J. (2010). 100 gigabit Ethernet and beyond [Commentary]. *IEEE Communications Magazine*, *48*(3), S6–S13. https://doi.org/10.1109/MCOM.2010.5434372 (see p. 30).

da Silva, S., Yemini, Y., & Florissi, D. (2001). The NetScript active network system. *IEEE J. Sel. Areas Commun.*, *19*(3), 538–551. https://doi.org/10.1109/49.917713 (see p. 15).

Dennard, R., Gaensslen, F., Yu, H.-N., Rideout, V., Bassous, E., & LeBlanc, A. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, *9*(5), 256–268. https://doi.org/10.1109/JSSC.1974.1050511 (see p. 29).

Deri, L. (2004). Improving Passive Packet Capture: Beyond Device Polling. In W. Belgers (Ed.), *Proceedings of SANE 2004, Amsterdam*. Retrieved March 18, 2022, from https://luca.ntop.org/Ring.pdf (see p. 41).

de Ruiter, J., & Schutijser, C. (2021). Next-generation internet at terabit speed: SCION in P4. In G. Carle & J. Ott (Eds.), *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021* (pp. 119–125). ACM. https://doi.org/10.1145/3485983.3494839 (see p. 28).

Dethise, A., Canini, M., & Kandula, S. (2019). Cracking Open the Black Box: What Observations Can Tell Us About Reinforcement Learning Agents. *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*, 29–36. https://doi.org/10.1145/3341216.3342210 (see p. 97).

Ding, D., Savi, M., Pederzolli, F., Campanella, M., & Siracusa, D. (2021). In-Network Volumetric DDoS Victim Identification Using Programmable Commodity Switches. *IEEE Trans. Netw. Serv. Manag.*, *18*(2), 1191–1202. https://doi.org/10.1109/TNSM.2021.3073597 (see pp. 51, 188).

Discord. (2022). *Discord: Your Place to Talk and Hang Out*. Retrieved April 21, 2022, from http://discord.com (see pp. 134, 223).

Doan, B. G., Abbasnejad, E., & Ranasinghe, D. C. (2020). Februus: Input Purification Defense Against Trojan Attacks on Deep Neural Network Systems. *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020*, 897–912. https://doi.org/10.1145/3427228.3427264 (see p. 104).

Dobbins, R., Migault, D., Moskowitz, R., Teague, N., Xia, L., & Nishizuka, K. (2021). Use Cases for DDoS Open Threat Signaling. https://doi.org/10.17487/RFC8903 (see pp. 117, 123).

Dong, M., Li, Q., Zarchy, D., Godfrey, P. B., & Schapira, M. (2015). PCC: Re-architecting Congestion Control for Consistent High Performance. *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, 395–408. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/dong (see pp. 53, 65).

Dong, M., Meng, T., Zarchy, D., Arslan, E., Gilad, Y., Godfrey, B., & Schapira, M. (2018). PCC Vivace: Online-Learning Congestion Control. In S. Banerjee & S. Seshan (Eds.), *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (pp. 343–356). USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/dong (see pp. 53, 65).

DPDK Project. (2022). *DPDK*. Retrieved March 17, 2022, from https://www.dpdk.org/ (see p. 42).

Draper-Gil, G., Lashkari, A. H., Mamun, M. S. I., & Ghorbani, A. A. (2016). Characterization of Encrypted and VPN Traffic using Time-related Features. In O. Camp, S. Furnell & P. Mori (Eds.), *Proceedings of the 2nd International Conference on Information Systems Security and Privacy, ICISSP 2016, Rome, Italy, February 19-21, 2016* (pp. 407–414). SciTePress. https://doi.org/10.5220/0005740704070414 (see p. 64).

Duarte, J., Harris, P., Hauck, S., Holzman, B., Hsu, S.-C., Jindariani, S., Khan, S., Kreis, B., Lee, B., Liu, M., Lončar, V., Ngadiuba, J., Pedro, K., Perez, B., Pierini, M., Rankin, D., Tran, N., Trahms, M., Tsaris, A., … Wu, Z. (2019). FPGA-Accelerated Machine Learning Inference as a Service for Particle Physics Computing. *Computing and Software for Big Science*, *3*(1), 13. https://doi.org/10.1007/s41781-019-0027-2 (see pp. 55, 147).

Duchi, J. C., Hazan, E., & Singer, Y. (2011). Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *J. Mach. Learn. Res.*, *12*, 2121–2159. http://dl.acm.org/citation.cfm?id=2021068 (see p. 84).

Dumitrescu, D., Stoenescu, R., Negreanu, L., & Raiciu, C. (2020). bf4: towards bug-free P4 programs. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 571–585). ACM. https://doi.org/10.1145/3387514.3405888 (see p. 33).

Edge, J. (2020, September 15). *BPF in GCC*. Retrieved March 25, 2022, from https://lwn.net/Articles/831402/ (see p. 45).

Eliyahu, T., Kazak, Y., Katz, G., & Schapira, M. (2021). Verifying learning-augmented systems. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM*

SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021 (pp. 305–318). ACM. https://doi.org/10.1145/3452296.3472936 (see p. 97).

Elwalid, A., Xiao, X., i_widjaja@yahoo.com, Chiu, A., & Awduche, D. O. (2002). Overview and Principles of Internet Traffic Engineering. https://doi.org/10.17487/RFC3272 (see p. 61).

ESnet. (2019, October 21). *ESnet HighTouch Telemetry 2019 Dataset*. Retrieved May 9, 2022, from https://downloads.es.net/pub/hightouch/imc2019/ (see p. 201).

Facebook Incubator. (2020). *Katran: A high performance layer 4 load balancer*. Retrieved March 3, 2022, from https://github.com/facebookincubator/katran (see pp. 48, 51).

Feamster, N., Balakrishnan, H., Rexford, J., Shaikh, A., & van der Merwe, J. (2004). The Case for Separating Routing from Routers. *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture*, 5–12. https://doi.org/10.1145/1016707.1016709 (see p. 19).

Feamster, N., & Rexford, J. (2018). Why (and How) Networks Should Run Themselves. *Proceedings of the Applied Networking Research Workshop, ANRW 2018, Montreal, QC, Canada, July 16-16, 2018*, 20. https://doi.org/10.1145/3232755.3234555 (see p. 59).

Feamster, N., Rexford, J., & Zegura, E. W. (2014). The road to SDN: an intellectual history of programmable networks. *Comput. Commun. Rev.*, *44*(2), 87–98. https://doi.org/10.1145/2602204.2602219 (see pp. 13, 14, 18, 29).

Feldmeier, D. C., McAuley, A. J., Smith, J. M., Bakin, D. S., Marcus, W. S., & Raleigh, T. (1998). Protocol boosters. *IEEE J. Sel. Areas Commun.*, *16*(3), 437–444. https://doi.org/10.1109/49.669053 (see p. 14).

Ferguson, A. D., Gribble, S. D., Hong, C., Killian, C. E., Mohsin, W., Mühe, H., Ong, J., Poutievski, L., Singh, A., Vicisano, L., Alimi, R., Chen, S. S., Conley, M., Mandal, S., Nagaraj, K., Bollineni, K. N., Sabaa, A., Zhang, S., Zhu, M., & Vahdat, A. (2021). Orion: Google's Software-Defined Networking Control Plane. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 83–98). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/ferguson (see p. 21).

Feurer, M., Springenberg, J. T., & Hutter, F. (2015). Initializing Bayesian Hyperparameter Optimization via Meta-Learning. In B. Bonet & S. Koenig (Eds.), *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA* (pp. 1128–1135). AAAI Press. http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/10029 (see p. 64).

Fleming, M. (2017, December 2). *A thorough introduction to eBPF*. Retrieved March 17, 2022, from https://lwn.net/Articles/740157/ (see p. 42).

Floyd, S., & Paxson, V. (2001). Difficulties in simulating the internet. *IEEE/ACM Trans. Netw.*, *9*(4), 392–403. https://doi.org/10.1109/90.944338 (see p. 94).

Fog, A. (2021). *Optimization manuals 4—Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*. Retrieved October 27, 2021, from https://www.agner.org/optimize/instruction_tables.pdf (see p. 91).

Fowers, J., Ovtcharov, K., Papamichael, M., Massengill, T., Liu, M., Lo, D., Alkalay, S., Haselman, M., Adams, L., Ghandi, M., Heil, S., Patel, P., Sapek, A., Weisz, G., Woods, L., Lanka, S., Reinhardt, S. K., Caulfield, A. M., Chung, E. S., & Burger, D. (2018). A Configurable Cloud-Scale DNN Processor for Real-Time AI. In M. Annavaram, T. M. Pinkston & B. Falsafi (Eds.), *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018* (pp. 1–14). IEEE Computer Society. https://doi.org/10.1109/ISCA.2018.00012 (see pp. 30, 55, 92, 147, 198).

Frazier, H. (1998). The 802.3z Gigabit Ethernet Standard. *IEEE Network*, *12*(3), 6–7. https://doi.org/10.1109/65.690946 (see p. 30).

Fredrikson, M., Jha, S., & Ristenpart, T. (2015). Model Inversion Attacks that Exploit Confidence Information and Basic Countermeasures. In I. Ray, N. Li & C. Kruegel (Eds.), *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015* (pp. 1322–1333). ACM. https://doi.org/10.1145/2810103.2813677 (see p. 105).

Fu, S., Gupta, S., Mittal, R., & Ratnasamy, S. (2021). On the Use of ML for Blackbox System Performance Prediction. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 763–784). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/fu (see p. 96).

Fujimoto, S., van Hoof, H., & Meger, D. (2018). Addressing Function Approximation Error in Actor-Critic Methods. In J. G. Dy & A. Krause (Eds.), *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018* (pp. 1582–1591, Vol. 80). PMLR. http://proceedings.mlr.press/v80/fujimoto18a.html (see p. 89).

Gail, R., & Kleinrock, L. (1981). An Invariant Property of Computer Network Power. *Proceedings of the International Conference on Communications. Denver, Colorado, 63.1.1–63.1.5* (see p. 66).

Gallo, M., & Laufer, R. P. (2018). ClickNF: a Modular Stack for Custom Network Functions. In H. S. Gunawi & B. Reed (Eds.), *2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018* (pp. 745–757). USENIX Association. https://www.usenix.org/conference/atc18/presentation/gallo (see p. 24).

Gao, J., Zhai, E., Liu, H. H., Miao, R., Zhou, Y., Tian, B., Sun, C., Cai, D., Zhang, M., & Yu, M. (2020). Lyra: A Cross-Platform Language and Compiler for Data Plane Programming on Heterogeneous ASICs. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 435–450). ACM. https://doi.org/10.1145/3387514.3405879 (see p. 32).

George, L., & Blume, M. (2003). Taming the IXP network processor. In R. Cytron & R. Gupta (Eds.), *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003, San Diego, California, USA, June 9-11, 2003* (pp. 26–37). ACM. https://doi.org/10.1145/781131.781135 (see p. 16).

Geyer, F., & Schmid, S. (2019). DeepMPLS: Fast Analysis of MPLS Configurations Using Deep Learning. *2019 IFIP Networking Conference, Networking 2019, Warsaw, Poland, May 20-22, 2019*, 1–9. https://doi.org/10.23919/IFIPNetworking.2019.8816842 (see p. 69).

Ghasemi, M., Benson, T., & Rexford, J. (2017). Dapper: Data Plane Performance Diagnosis of TCP. *Proceedings of the Symposium on SDN Research, SOSR 2017, Santa Clara, CA, USA, April 3-4, 2017*, 61–74. https://doi.org/10.1145/3050220.3050228 (see pp. 49, 188).

Gigis, P., Calder, M., Manassakis, L., Nomikos, G., Kotronis, V., Dimitropoulos, X. A., Katz-Bassett, E., & Smaragdakis, G. (2021). Seven years in the life of Hypergiants' off-nets. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021* (pp. 516–533). ACM. https://doi.org/10.1145/3452296.3472928 (see p. 30).

Gilad, T., Schiff, N. R., Godfrey, P. B., Raiciu, C., & Schapira, M. (2020). MPCC: online learning multipath transport. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 121–135). ACM. https://doi.org/10.1145/3386367.3433030 (see p. 65).

Giladi, R. (2008). *Network Processors: Architecture, Programming, and Implementation* (W. Wolf, Ed.). Morgan Kaufmann. Retrieved February 9, 2022, from https://web.archive.org/web/20220209095706/https://doc.lagout.org/network/Network%20Processors%20-%20Architecture%2C%20Programming%2C%20and%20Implementation.pdf (see pp. 25, 26).

Gnanasambandam, A., Sherman, A. M., & Chan, S. H. (2021). Optical Adversarial Attack. *CoRR, abs/2108.06247*. https://arxiv.org/abs/2108.06247 (see p. 99).

Goodfellow, I. J., Bengio, Y., & Courville, A. C. (2016). *Deep Learning* (T. Dietterich, Ed.). MIT Press. http://www.deeplearningbook.org/ (see pp. 81, 84).

Goodfellow, I. J., Shlens, J., & Szegedy, C. (2014). Explaining and Harnessing Adversarial Examples. *CoRR, abs/1412.6572*. http://arxiv.org/abs/1412.6572 (see p. 99).

Grounds, M. J., & Kudenko, D. (2007). Parallel Reinforcement Learning with Linear Function Approximation. In K. Tuyls, A. Nowé, Z. Guessoum & D. Kudenko (Eds.), *Adaptive Agents and Multi-Agent Systems III. Adaptation and Multi-Agent Learning, 5th, 6th, and 7th European Symposium, ALAMAS 2005-2007 on Adaptive and Learning Agents and Multi-Agent Systems, Revised Selected Papers* (pp. 60–74, Vol. 4865). Springer. https://doi.org/10.1007/978-3-540-77949-0_5 (see pp. 150, 152).

Guan, Y., Zhang, Y., Wang, B., Bian, K., Xiong, X., & Song, L. (2020). PERM: Neural Adaptive Video Streaming with Multi-path Transmission. *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, 1103–1112. https://doi.org/10.1109/INFOCOM41043.2020.9155492 (see p. 70).

Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., & Shenker, S. (2008). NOX: towards an operating system for networks. *Comput. Commun. Rev.*, *38*(3), 105–110. https://doi.org/10.1145/1384609.1384625 (see p. 20).

Guéant, V. (2020). *iPerf – The TCP, UDP and SCTP network bandwidth measurement tool*. Retrieved May 3, 2022, from https://iperf.fr/ (see p. 201).

Guok, C., Cziva, R., Kumar, Y., & Mah, B. A. (2021, August 4). *ESnet6 High-Touch Platform: Precision Streaming Network Telemetry* [Presented at the 2nd Global Research Platform Workshop.]. Retrieved March 10, 2022, from https://grpworkshop2021.theglobalresearchplatform.net/PDF/5-GUOK-GRP-2021-ESnet6-High%20Touch.pdf (see p. 29).

Gupta, A., Harrison, R., Canini, M., Feamster, N., Rexford, J., & Willinger, W. (2018). Sonata: query-driven streaming network telemetry. In S. Gorinsky & J. Tapolcai (Eds.), *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (pp. 357–371). ACM. https://doi.org/10.1145/3230543.3230555 (see pp. 49, 64, 188, 207).

Gurtov, A., Henderson, T., Floyd, S., & Nishida, Y. (2012). The NewReno Modification to TCP's Fast Recovery Algorithm. https://doi.org/10.17487/RFC6582 (see p. 2).

Hagos, D. H., Engelstad, P. E., Yazidi, A., & Kure, Ø. (2018). Towards a Robust and Scalable TCP Flavors Prediction Model from Passive Traffic. *27th International Conference on Computer Communication and Networks, ICCCN 2018, Hangzhou, China, July 30 - August 2, 2018*, 1–11. https://doi.org/10.1109/ICCCN.2018.8487396 (see pp. 64, 201, 204).

Hamming, R. W. (1997). *The Art of Doing Science and Engineering: Learning to Learn* (4th ed.). Stripe Press. (See p. 95).

Han, Y., Hubczenko, D., Montague, P., de Vel, O. Y., Abraham, T., Rubinstein, B. I. P., Leckie, C., Alpcan, T., & Erfani, S. M. (2019). Adversarial Reinforcement Learning under Partial Observability in Software-Defined Networking. *CoRR, abs/1902.09062*. http://arxiv.org/abs/1902.09062 (see p. 119).

Handigol, N., Heller, B., Jeyakumar, V., Mazières, D., & McKeown, N. (2014). I Know What Your Packet Did Last Hop: Using Packet Histories to Troubleshoot Networks. In R. Mahajan & I. Stoica (Eds.), *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (pp. 71–85). USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/handigol (see p. 50).

Handley, M., Hodson, O., & Kohler, E. (2003). XORP: an open platform for network research. *Comput. Commun. Rev.*, *33*(1), 53–57. https://doi.org/10.1145/774763.774771 (see p. 19).

Handley, M., Raiciu, C., Agache, A., Voinescu, A., Moore, A. W., Antichi, G., & Wójcik, M. (2017). Re-architecting datacenter networks and stacks for low latency and high performance. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 29–42. https://doi.org/10.1145/3098822.3098825 (see p. 53).

Harary, F., & Norman, R. Z. (1960). Some properties of line digraphs. *Rendiconti del Circolo Matematico di Palermo*, *9*(2), 161–168. https://doi.org/10.1007/BF02854581 (see p. 73).

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, 770–778. https://doi.org/10.1109/CVPR.2016.90 (see p. 3).

He, Y., Meng, G., Chen, K., Hu, X., & He, J. (2021). DRMI: A Dataset Reduction Technology based on Mutual Information for Black-box Attacks. In M. Bailey & R. Greenstadt (Eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (pp. 1901–1918). USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/he-yingzhe (see p. 105).

Heart, F. E., Kahn, R. E., Ornstein, S. M., Crowther, W. R., & Walden, D. C. (1970). The interface message processor for the ARPA computer network. In H. L. Cooke (Ed.), *American Federation of Information Processing Societies: AFIPS Conference Proceedings: 1970 Spring Joint Computer Conference, Atlantic City, NJ, USA, May 5-7, 1970* (pp. 551–567, Vol. 36). AFIPS Press. https://doi.org/10.1145/1476936.1477021 (see p. 1).

Hilton, S. (2016, October 26). *Dyn Analysis Summary Of Friday October 21 Attack*. Dyn. Retrieved April 2, 2022, from https://web.archive.org/web/20200227230110/https://dyn.com/blog/dyn-analysis-summary-of-friday-october-21-attack/ (see p. 111).

Hinton, G. E., Vinyals, O., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. *CoRR, abs/1503.02531*. http://arxiv.org/abs/1503.02531 (see p. 100).

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation, 9*(8), 1735–1780. https://doi.org/10.1162/neco.1997.9.8.1735 (see p. 82).

Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., & Miller, D. (2018). The eXpress data path: fast programmable packet processing in the operating system kernel. In X. A. Dimitropoulos, A. Dainotti, L. Vanbever & T. Benson (Eds.), *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018* (pp. 54–66). ACM. https://doi.org/10.1145/3281411.3281443 (see p. 43).

Hong, C., Mandal, S., Al-Fares, M., Zhu, M., Alimi, R., Bollineni, K. N., Bhagat, C., Jain, S., Kaimal, J., Liang, S., Mendelev, K., Padgett, S., Rabe, F., Ray, S., Tewari, M., Tierney, M., Zahn, M., Zolla, J., Ong, J., & Vahdat, A. (2018). B4 and after: managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN. In S. Gorinsky & J. Tapolcai (Eds.), *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (pp. 74–87). ACM. https://doi.org/10.1145/3230543.3230545 (see p. 21).

Hsu, K., Beckett, R., Chen, A., Rexford, J., & Walker, D. (2020). Contra: A Programmable System for Performance-aware Routing. In R. Bhagwan & G. Porter (Eds.), *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (pp. 701–721). USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/hsu (see p. 52).

Hu, S., Chen, K., Wu, H., Bai, W., Lan, C., Wang, H., Zhao, H., & Guo, C. (2016). Explicit Path Control in Commodity Data Centers: Design and Applications. *IEEE/ACM Trans. Netw., 24*(5), 2768–2781. https://doi.org/10.1109/TNET.2015.2482988 (see p. 62).

Hu, Z., Li, D., Zhang, D., & Chen, Y. (2020). ReLoca: Optimize Resource Allocation for Data-parallel Jobs using Deep Learning. *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, 1163–1171. https://doi.org/10.1109/INFOCOM41043.2020.9155521 (see p. 72).

Huang, S. H., Papernot, N., Goodfellow, I. J., Duan, Y., & Abbeel, P. (2017). Adversarial Attacks on Neural Network Policies. *CoRR, abs/1702.02284*. http://arxiv.org/abs/1702.02284 (see pp. 100, 118).

Huang, T., Zhou, C., Zhang, R., Wu, C., Yao, X., & Sun, L. (2020). Stick: A Harmonious Fusion of Buffer-based and Learning-based Approach for Adaptive Streaming. *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, 1967–

1976. https://doi.org/10.1109/INFOCOM41043.2020.9155411 (see p. 70).

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., & Bengio, Y. (2016). Binarized Neural Networks. In D. D. Lee, M. Sugiyama, U. von Luxburg, I. Guyon & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain* (pp. 4107–4115). http://papers.nips.cc/paper/6573-binarized-neural-networks (see p. 93).

Hügerich, L., Shukla, A., & Smaragdakis, G. (2021). No-hop: In-network Distributed Hash Tables. *ANCS '21: Symposium on Architectures for Networking and Communications Systems, Layfette, IN, USA, December 13 - 16, 2021*, 80–87. https://doi.org/10.1145/3493425.3502757 (see p. 51).

Hutter, F., Hoos, H. H., & Leyton-Brown, K. (2011). Sequential Model-Based Optimization for General Algorithm Configuration. In C. A. C. Coello (Ed.), *Learning and Intelligent Optimization - 5th International Conference, LION 5, Rome, Italy, January 17-21, 2011. Selected Papers* (pp. 507–523, Vol. 6683). Springer. https://doi.org/10.1007/978-3-642-25566-3_40 (see p. 64).

Hypolite, J., Sonchack, J., Hershkop, S., Dautenhahn, N., DeHon, A., & Smith, J. M. (2020). DeepMatch: practical deep packet inspection in the data plane using network processors. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 336–350). ACM. https://doi.org/10.1145/3386367.3431290 (see p. 52).

Ibanez, S., Antichi, G., Brebner, G. J., & McKeown, N. (2019). Event-Driven Packet Processing. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019* (pp. 133–140). ACM. https://doi.org/10.1145/3365609.3365848 (see pp. 32, 198).

Ibanez, S., Brebner, G. J., McKeown, N., & Zilberman, N. (2019). The P4->NetFPGA Workflow for Line-Rate Packet Processing. In K. Bazargan & S. Neuendorffer (Eds.), *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019* (pp. 1–9). ACM. https://doi.org/10.1145/3289602.3293924 (see pp. 28, 75).

Ibanez, S., Mallery, A., Arslan, S., Jepsen, T., Shahbaz, M., Kim, C., & McKeown, N. (2021). The nanoPU: A Nanosecond Network Stack for Datacenters. In A. D. Brown & J. R. Lorch (Eds.), *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021* (pp. 239–256). USENIX Association. https://www.usenix.org/conference/osdi21/presentation/ibanez (see p. 35).

Ilyas, A., Jalal, A., Asteri, E., Daskalakis, C., & Dimakis, A. G. (2017). The Robust Manifold Defense: Adversarial Training using Generative

Models. *CoRR, abs/1712.09196*. http://arxiv.org/abs/1712.09196 (see p. 101).

Informa Tech. (2006, January 23). *Broadcom Pays $80 Million For Sandburst*. Retrieved March 4, 2022, from https://www.networkcomputing.com/networking/broadcom-pays-80-million-sandburst (see p. 25).

Intel. (2001). *Intel® IXP1200 Network Processor*. Retrieved February 7, 2022, from https://web.archive.org/web/20220207202018/https://www.marutsu.co.jp/contents/shop/marutsu/ds/IXP1200.pdf (see pp. 16, 25).

Intel. (2017). *Intel Data Direct I/O Technology*. Retrieved March 16, 2022, from https://www.intel.co.uk/content/www/uk/en/io/data-direct-i-o-technology.html (see p. 40).

Intel. (2021a). *3rd Gen Intel® Xeon® Scalable Processors Brief*. Retrieved September 27, 2021, from https://www.intel.co.uk/content/www/uk/en/products/docs/processors/xeon/3rd-gen-xeon-scalable-processors-brief.html (see p. 92).

Intel. (2021b, June 14). *Intel Unveils Infrastructure Processing Unit*. Retrieved March 10, 2022, from https://www.intel.com/content/www/us/en/newsroom/news/infrastructure-processing-unit-data-center.html#:~:text=An%20IPU%20(infrastructure%20processing%20unit,data%20center%20with%20programmable%20hardware (see pp. 2, 32).

Intel. (2022). *Intel Tofino 2: P4 Programmability with More Bandwidth*. Retrieved March 4, 2022, from https://www.intel.co.uk/content/www/uk/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html (see pp. 2, 31).

Intel Networking Division. (2017). *Intel Ethernet Switch FM5000/FM6000: Datasheet*. Retrieved February 24, 2022, from https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/ethernet-switch-fm5000-fm6000-datasheet.pdf (see p. 27).

Iordache, C., & Tang, P. T. P. (2003). An Overview of Floating-Point Support and Math Library on the Intel XScale^{TM} Architecture. *16th IEEE Symposium on Computer Arithmetic (Arith-16 2003), 15-18 June 2003, Santiago de Compostela, Spain*, 122–128. https://doi.org/10.1109/ARITH.2003.1207669 (see p. 154).

Iordache-Şică, M. M., Anagnostopoulos, C., & Pezaros, D. P. (2021). Towards QoS-aware Provisioning of Chained Virtual Security Services in Edge Networks. In T. Ahmed, O. Festor, Y. Ghamri-Doudane, J. Kang, A. E. S. Filho, A. Lahmadi & E. R. M. Madeira (Eds.), *17th IFIP/IEEE International Symposium on Integrated Network Management, IM 2021, Bordeaux, France, May 17-21, 2021* (pp. 178–186). IEEE. https://ieeexplore.ieee.org/document/9464064 (see p. 24).

Jagielski, M., Carlini, N., Berthelot, D., Kurakin, A., & Papernot, N. (2020). High Accuracy and High Fidelity Extraction of Neural Networks. In S. Capkun & F. Roesner (Eds.), *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020* (pp. 1345–1362).

USENIX Association. https://www.usenix.org/conference/usenixsecurity20/presentation/jagielski (see pp. 104, 105).

Jain, S., Kumar, A., Mandal, S., Ong, J., Poutievski, L., Singh, A., Venkata, S., Wanderer, J., Zhou, J., Zhu, M., Zolla, J., Hölzle, U., Stuart, S., & Vahdat, A. (2013). B4: experience with a globally-deployed software defined wan. In D. M. Chiu, J. Wang, P. Barford & S. Seshan (Eds.), *ACM SIGCOMM 2013 Conference, SIGCOMM 2013, Hong Kong, August 12-16, 2013* (pp. 3–14). ACM. https://doi.org/10.1145/2486001.2486019 (see p. 21).

Jakaria, A. H. M., Rahman, M. A., & Fung, C. J. (2019). A Requirement-Oriented Design of NFV Topology by Formal Synthesis. *IEEE Trans. Netw. Serv. Manag.*, *16*(4), 1739–1753. https://doi.org/10.1109/TNSM.2019.2920824 (see p. 119).

Jay, N., Rotman, N. H., Godfrey, B., Schapira, M., & Tamar, A. (2019). A Deep Reinforcement Learning Perspective on Internet Congestion Control. In K. Chaudhuri & R. Salakhutdinov (Eds.), *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA* (pp. 3050–3059, Vol. 97). PMLR. http://proceedings.mlr.press/v97/jay19a.html (see p. 66).

Jayaraman, B., & Evans, D. (2019). Evaluating Differentially Private Machine Learning in Practice. In N. Heninger & P. Traynor (Eds.), *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019* (pp. 1895–1912). USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/jayaraman (see p. 106).

Jensen, J. S., Krøgh, T. B., Madsen, J. S., Schmid, S., Srba, J., & Thorgersen, M. T. (2018). P-Rex: fast verification of MPLS networks with multiple link failures. In X. A. Dimitropoulos, A. Dainotti, L. Vanbever & T. Benson (Eds.), *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018* (pp. 217–227). ACM. https://doi.org/10.1145/3281411.3281432 (see p. 69).

Jepsen, T., Fattaholmanan, A., Moshref, M., Foster, N., Carzaniga, A., & Soulé, R. (2020). Forwarding and routing with packet subscriptions. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 282–294). ACM. https://doi.org/10.1145/3386367.3431315 (see p. 51).

Jeyakumar, V., Alizadeh, M., Geng, Y., Kim, C., & Mazières, D. (2014). Millions of little minions: using packets for low latency network programming and visibility. In F. E. Bustamante, Y. C. Hu, A. Krishnamurthy & S. Ratnasamy (Eds.), *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014* (pp. 3–14). ACM. https://doi.org/10.1145/2619239.2626292 (see pp. 22, 50).

Jiang, J., Sekar, V., Stoica, I., & Zhang, H. (2017). Unleashing the Potential of Data-Driven Networking. In N. R. Sastry & S. Chakraborty

(Eds.), *Communication Systems and Networks - 9th International Conference, COMSNETS 2017, Bengaluru, India, January 4-8, 2017, Revised Selected Papers and Invited Papers* (pp. 110–126, Vol. 10340). Springer. https://doi.org/10.1007/978-3-319-67235-9_9 (see p. 60).

Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C., & Stoica, I. (2018). NetChain: Scale-Free Sub-RTT Coordination. In S. Banerjee & S. Seshan (Eds.), *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (pp. 35–49). USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/jin (see p. 51).

Jin, X., Li, X., Zhang, H., Soulé, R., Lee, J., Foster, N., Kim, C., & Stoica, I. (2017). NetCache: Balancing Key-Value Stores with Fast In-Network Caching. *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*, 121–136. https://doi.org/10.1145/3132747.3132764 (see p. 193).

Jog, S., Liu, Z., Franques, A., Fernando, V., Abadal, S., Torrellas, J., & Hassanieh, H. (2021). One Protocol to Rule Them All: Wireless Network-on-Chip using Deep Reinforcement Learning. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 973–989). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/jog (see p. 66).

Jonker, M., King, A., Krupp, J., Rossow, C., Sperotto, A., & Dainotti, A. (2017). Millions of targets under attack: a macroscopic characterization of the DoS ecosystem. In S. Uhlig & O. Maennel (Eds.), *Proceedings of the 2017 Internet Measurement Conference, IMC 2017, London, United Kingdom, November 1-3, 2017* (pp. 100–113). ACM. https://doi.org/10.1145/3131365.3131383 (see pp. 111–113, 115).

Jouët, S., Perkins, C., & Pezaros, D. P. (2016). OTCP: SDN-managed congestion control for data center networks. In S. Oktug, M. Ulema, C. Cavdar, L. Z. Granville & C. R. P. dos Santos (Eds.), *2016 IEEE/IFIP Network Operations and Management Symposium, NOMS 2016, Istanbul, Turkey, April 25-29, 2016* (pp. 171–179). IEEE. https://doi.org/10.1109/NOMS.2016.7502810 (see p. 53).

Jouët, S., & Pezaros, D. P. (2017). BPFabric: Data Plane Programmability for Software Defined Networks. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems, ANCS 2017, Beijing, China, May 18-19, 2017* (pp. 38–48). IEEE. https://doi.org/10.1109/ANCS.2017.14 (see p. 47).

Juuti, M., Szyller, S., Marchal, S., & Asokan, N. (2019). PRADA: Protecting Against DNN Model Stealing Attacks. *IEEE European Symposium on Security and Privacy, EuroS&P 2019, Stockholm, Sweden, June 17-19, 2019*, 512–527. https://doi.org/10.1109/EuroSP.2019.00044 (see p. 106).

Kalia, A., Kaminsky, M., & Andersen, D. G. (2019). Datacenter RPCs can be General and Fast. In J. R. Lorch & M. Yu (Eds.), *16th USENIX*

Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019 (pp. 1–16). USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/kalia (see p. 39).

Kamp, P.-H., & Watson, R. N. M. (2000). *Jails: Confining the omnipotent root.* Retrieved February 28, 2022, from https://papers.freebsd.org/2000/phk-jails/ (see p. 24).

Kang, M. S., Gligor, V. D., & Sekar, V. (2016a). Defending Against Evolving DDoS Attacks: A Case Study Using Link Flooding Incidents. In J. Anderson, V. Matyás, B. Christianson & F. Stajano (Eds.), *Security Protocols XXIV - 24th International Workshop, Brno, Czech Republic, April 7-8, 2016, Revised Selected Papers* (pp. 47–57, Vol. 10368). Springer. https://doi.org/10.1007/978-3-319-62033-6_7 (see pp. 111, 212).

Kang, M. S., Gligor, V. D., & Sekar, V. (2016b). SPIFFY: Inducing Cost-Detectability Tradeoffs for Persistent Link-Flooding Attacks. *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/spiffy-inducing-cost-detectability-tradeoffs-persistent-link-flooding-attacks.pdf (see pp. 68, 110, 115, 116, 118, 119, 124, 135, 188).

Kang, M. S., Lee, S. B., & Gligor, V. D. (2013). The Crossfire Attack. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013* (pp. 127–141). IEEE Computer Society. https://doi.org/10.1109/SP.2013.19 (see p. 114).

Kansal, N., Ramanujam, M., & Netravali, R. (2021). Alohamora: Reviving HTTP/2 Push and Preload by Adapting Policies On the Fly. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 269–287). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/kansal (see p. 71).

Katsikas, G. P., Barbette, T., Kostic, D., Steinert, R., & Jr., G. Q. M. (2018). Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In S. Banerjee & S. Seshan (Eds.), *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (pp. 171–186). USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/katsikas (see p. 48).

Katta, N. P., Hira, M., Kim, C., Sivaraman, A., & Rexford, J. (2016). HULA: Scalable Load Balancing Using Programmable Data Planes. In B. Godfrey & M. Casado (Eds.), *Proceedings of the Symposium on SDN Research, SOSR 2016, Santa Clara, CA, USA, March 14 - 15, 2016* (p. 10). ACM. https://doi.org/10.1145/2890955.2890968 (see p. 52).

Katz, G., Barrett, C. W., Dill, D. L., Julian, K., & Kochenderfer, M. J. (2017). Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In R. Majumdar & V. Kuncak (Eds.), *Computer*

Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (pp. 97–117, Vol. 10426). Springer. https://doi.org/10.1007/978-3-319-63387-9_5 (see p. 97).

Katz, G., Huang, D. A., Ibeling, D., Julian, K., Lazarus, C., Lim, R., Shah, P., Thakoor, S., Wu, H., Zeljic, A., Dill, D. L., Kochenderfer, M. J., & Barrett, C. W. (2019). The Marabou Framework for Verification and Analysis of Deep Neural Networks. In I. Dillig & S. Tasiran (Eds.), *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I* (pp. 443–452, Vol. 11561). Springer. https://doi.org/10.1007/978-3-030-25540-4_26 (see p. 97).

Kazak, Y., Barrett, C. W., Katz, G., & Schapira, M. (2019). Verifying Deep-RL-Driven Systems. *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*, 83–89. https://doi.org/10.1145/3341216.3342218 (see p. 97).

Kellerer, W., Kalmbach, P., Blenk, A., Basta, A., Reisslein, M., & Schmid, S. (2019). Adaptable and Data-Driven Softwarized Networks: Review, Opportunities, and Challenges. *Proc. IEEE*, *107*(4), 711–731. https://doi.org/10.1109/JPROC.2019.2895553 (see p. 59).

Kesarwani, M., Mukhoty, B., Arya, V., & Mehta, S. (2018). Model Extraction Warning in MLaaS Paradigm. *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*, 371–380. https://doi.org/10.1145/3274694.3274740 (see p. 106).

Keutzer, K., Malik, S., & Newton, A. R. (2002). From ASIC to ASIP: The Next Design Discontinuity. *20th International Conference on Computer Design (ICCD 2002), VLSI in Computers and Processors, 16-18 September 2002, Freiburg, Germany, Proceedings*, 84–90. https://doi.org/10.1109/ICCD.2002.1106752 (see p. 25).

Kfoury, E. F., Crichigno, J., & Bou-Harb, E. (2021). An Exhaustive Survey on P4 Programmable Data Plane Switches: Taxonomy, Applications, Challenges, and Future Trends. *IEEE Access*, *9*, 87094–87155. https://doi.org/10.1109/ACCESS.2021.3086704 (see p. 49).

Kim, J., Jung, Y., Yeo, H., Ye, J., & Han, D. (2020). Neural-Enhanced Live Streaming: Improving Live Video Ingest via Online Learning. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 107–125). ACM. https://doi.org/10.1145/3387514.3405856 (see p. 70).

Kim, M., & Smaragdis, P. (2016). Bitwise Neural Networks. *CoRR*, *abs/1601.06071*. http://arxiv.org/abs/1601.06071 (see p. 93).

Kingma, D. P., & Ba, J. (2014). Adam: A Method for Stochastic Optimization. *CoRR*, *abs/1412.6980*. http://arxiv.org/abs/1412.6980 (see pp. 84, 99).

Kipf, T. N., & Welling, M. (2017). Semi-Supervised Classification with Graph Convolutional Networks. *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. https://openreview.net/forum?id=SJU4ayYgl (see p. 81).

Kirilin, V., Sundarrajan, A., Gorinsky, S., & Sitaraman, R. K. (2020). RL-Cache: Learning-Based Cache Admission for Content Delivery. *IEEE J. Sel. Areas Commun.*, *38*(10), 2372–2385. https://doi.org/10.1109/JSAC.2020.3000415 (see p. 72).

Kivity, A., Kamay, Y., Laor, D., Lublin, U., & Liguori, A. (2007). kvm: the Linux Virtual Machine Monitor. *In Proceedings of the 2007 Ottawa Linux Symposium (OLS'-07)*, 225–230. Retrieved February 28, 2022, from https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf (see p. 24).

Klaba, O. (2016, September 22). OVH. Retrieved May 4, 2018, from https://twitter.com/olesovhcom/status/778830571677978624 (see p. 111).

Kleinrock, L. (1978). On Flow Control in Computer Networks. *Proceedings of the International Conference on Communications, Vol. 2. 27–2* (see p. 66).

Kloft, M., & Laskov, P. (2010). Online Anomaly Detection under Adversarial Impact. In Y. W. Teh & D. M. Titterington (Eds.), *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010* (pp. 405–412, Vol. 9). JMLR.org. http://www.jmlr.org/proceedings/papers/v9/kloft10a.html (see p. 103).

Koponen, T., Casado, M., Gude, N., Stribling, J., Poutievski, L., Zhu, M., Ramanathan, R., Iwata, Y., Inoue, H., Hama, T., & Shenker, S. (2010). Onix: A Distributed Control Platform for Large-scale Production Networks. In R. H. Arpaci-Dusseau & B. Chen (Eds.), *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings* (pp. 351–364). USENIX Association. http://www.usenix.org/events/osdi10/tech/full_papers/Koponen.pdf (see p. 20).

Kottler, S. (2018, March 1). *February 28th DDoS Incident Report*. GitHub Engineering. Retrieved May 4, 2018, from https://githubengineering.com/ddos-incident-report/ (see p. 111).

Krämer, L., Krupp, J., Makita, D., Nishizoe, T., Koide, T., Yoshioka, K., & Rossow, C. (2015). AmpPot: Monitoring and Defending Against Amplification DDoS Attacks. In H. Bos, F. Monrose & G. Blanc (Eds.), *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings* (pp. 615–636, Vol. 9404). Springer. https://doi.org/10.1007/978-3-319-26362-5_28 (see pp. 116, 132).

Krebs, B. (2016, September 21). *KrebsOnSecurity Hit With Record DDoS*. KrebsOnSecurity. Retrieved May 4, 2018, from https://krebsonsecurity.com/2016/09/krebsonsecurity-hit-with-record-ddos/ (see p. 111).

Kucera, J., Basat, R. B., Kuka, M., Antichi, G., Yu, M., & Mitzenmacher, M. (2020). Detecting routing loops in the data plane. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 466–473). ACM. https://doi.org/10.1145/3386367.3431303 (see p. 51).

Kührer, M., Hupperich, T., Bushart, J., Rossow, C., & Holz, T. (2015). Going Wild: Large-Scale Classification of Open DNS Resolvers. In K. Cho, K. Fukuda, V. S. Pai & N. Spring (Eds.), *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015* (pp. 355–368). ACM. https://doi.org/10.1145/2815675.2815683 (see p. 113).

Kührer, M., Hupperich, T., Rossow, C., & Holz, T. (2014). Exit from Hell? Reducing the Impact of Amplification DDoS Attacks. In K. Fu & J. Jung (Eds.), *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014.* (pp. 111–125). USENIX Association. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/kuhrer (see p. 113).

Kundel, R., Siegmund, F., Blendin, J., Rizk, A., & Koldehofe, B. (2020). P4STA: High Performance Packet Timestamping with Programmable Packet Processors. *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*, 1–9. https://doi.org/10.1109/NOMS47738.2020.9110290 (see p. 199).

Kunze, I., Gunz, M., Saam, D., Wehrle, K., & Rüth, J. (2021). Tofino + P4: A Strong Compound for AQM on High-Speed Networks? In T. Ahmed, O. Festor, Y. Ghamri-Doudane, J. Kang, A. E. S. Filho, A. Lahmadi & E. R. M. Madeira (Eds.), *17th IFIP/IEEE International Symposium on Integrated Network Management, IM 2021, Bordeaux, France, May 17-21, 2021* (pp. 72–80). IEEE. https://ieeexplore.ieee.org/document/9463943 (see p. 53).

Kurakin, A., Goodfellow, I. J., & Bengio, S. (2016). Adversarial examples in the physical world. *CoRR, abs/1607.02533*. http://arxiv.org/abs/1607.02533 (see p. 99).

Lakshman, T. V., Nandagopal, T., Ramjee, R., Sabnani, K., & Woo, T. (2004). The SoftRouter Architecture (ACM HOTNETS). *ACM HOTNETS*. Retrieved February 11, 2022, from https://www.microsoft.com/en-us/research/publication/the-softrouter-architecture/ (see p. 19).

Langlet, J. (2019). *Towards Machine Learning Inference in the Data Plane* [Bachelor's Thesis]. Karlstad University. Retrieved May 4, 2021, from http://urn.kb.se/resolve?urn=urn:nbn:se:kau:diva-72875 (see pp. 55, 93, 187, 228).

Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J. R., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., … Shi, Z. (2017). The QUIC Transport

Protocol: Design and Internet-Scale Deployment. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017* (pp. 183–196). ACM. https://doi.org/10.1145/3098822.3098842 (see pp. 22, 114, 115).

Lantz, B., Heller, B., & McKeown, N. (2010). A network in a laptop: rapid prototyping for software-defined networks. In G. G. Xie, R. Beverly, R. T. Morris & B. Davie (Eds.), *Proceedings of the 9th ACM Workshop on Hot Topics in Networks. HotNets 2010, Monterey, CA, USA - October 20 - 21, 2010* (p. 19). ACM. https://doi.org/10.1145/1868447.1868466 (see p. 21).

Lao, C., Le, Y., Mahajan, K., Chen, Y., Wu, W., Akella, A., & Swift, M. M. (2021). ATP: In-network Aggregation for Multi-tenant Learning. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 741–761). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/lao (see pp. 56, 93).

Larabel, M. (2015, January 24). *BPF Backend Merged Into LLVM To Make Use Of New Kernel Functionality*. Retrieved March 25, 2022, from https://www.phoronix.com/scan.php?page=news_item&px=LLVM-BPF-VM-Backend-Lands (see p. 45).

LeCun, Y. (1989). *Generalization and Network Design Strategies* (Technical Report No. CRG-TR-89-4). Department of Computer Science, University of Toronto. Retrieved November 1, 2021, from http://yann.lecun.com/exdb/publis/pdf/lecun-89.pdf (see p. 81).

Lécuyer, M., Atlidakis, V., Geambasu, R., Hsu, D., & Jana, S. (2019). Certified Robustness to Adversarial Examples with Differential Privacy. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 656–672. https://doi.org/10.1109/SP.2019.00044 (see p. 101).

Lécuyer, M., Lockerman, J., Nelson, L., Sen, S., Sharma, A., & Slivkins, A. (2017). Harvesting Randomness to Optimize Distributed Systems. In S. Banerjee, B. Karp & M. Walfish (Eds.), *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017* (pp. 178–184). ACM. https://doi.org/10.1145/3152434.3152435 (see p. 95).

Lee, S., Kim, J., Shin, S., Porras, P. A., & Yegneswaran, V. (2017). Athena: A Framework for Scalable Anomaly Detection in Software-Defined Networks. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2017, Denver, CO, USA, June 26-29, 2017* (pp. 249–260). IEEE Computer Society. https://doi.org/10.1109/DSN.2017.42 (see pp. 68, 116, 118).

Leland, W. E., Willinger, W., Taqqu, M. S., & Wilson, D. V. (1995). On the self-similar nature of Ethernet traffic. *Computer Communication Review*, *25*(1), 202–213. https://doi.org/10.1145/205447.205464 (see p. 67).

Li, B., Tan, K., Luo, L. L., Peng, Y., Luo, R., Xu, N., Xiong, Y., & Cheng, P. (2016). ClickNP: Highly flexible and High-performance Network Processing with Reconfigurable Hardware. In M. P. Barcellos, J. Crowcroft, A. Vahdat & S. Katti (Eds.), *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (pp. 1–14). ACM. https://doi.org/10.1145/2934872.2934897 (see p. 48).

Li, R., Xiao, X., Ni, S., Zheng, H., & Xia, S. (2018). Byte Segment Neural Network for Network Traffic Classification. *26th IEEE/ACM International Symposium on Quality of Service, IWQoS 2018, Banff, AB, Canada, June 4-6, 2018*, 1–10. https://doi.org/10.1109/IWQoS.2018.8624128 (see p. 64).

Li, Y., Liu, I., Yuan, Y., Chen, D., Schwing, A. G., & Huang, J. (2019). Accelerating distributed reinforcement learning with in-switch computing. In S. B. Manne, H. C. Hunter & E. R. Altman (Eds.), *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019* (pp. 279–291). ACM. https://doi.org/10.1145/3307650.3322259 (see pp. 56, 169, 200).

Li, Y., Miao, R., Liu, H. H., Zhuang, Y., Feng, F., Tang, L., Cao, Z., Zhang, M., Kelly, F., Alizadeh, M., & Yu, M. (2019). HPCC: high precision congestion control. In J. Wu & W. Hall (Eds.), *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (pp. 44–58). ACM. https://doi.org/10.1145/3341302.3342085 (see p. 53).

Liang, E., Zhu, H., Jin, X., & Stoica, I. (2019). Neural packet classification. In J. Wu & W. Hall (Eds.), *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (pp. 256–269). ACM. https://doi.org/10.1145/3341302.3342221 (see p. 63).

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., & Wierstra, D. (2016). Continuous control with deep reinforcement learning. In Y. Bengio & Y. LeCun (Eds.), *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*. http://arxiv.org/abs/1509.02971 (see p. 89).

Lin, J., Patel, K., Stephens, B. E., Sivaraman, A., & Akella, A. (2020). PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*, 243–259. https://www.usenix.org/conference/osdi20/presentation/lin (see p. 34).

Liu, G., Sadok, H., Kohlbrenner, A., Parno, B., Sekar, V., & Sherry, J. (2021). Don't Yank My Chain: Auditable NF Service Chaining. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14,*

2021 (pp. 155–173). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/liu-guyue (see p. 25).

Liu, J., Hallahan, W. T., Schlesinger, C., Sharif, M., Lee, J., Soulé, R., Wang, H., Cascaval, C., McKeown, N., & Foster, N. (2018). p4v: practical verification for programmable data planes. In S. Gorinsky & J. Tapolcai (Eds.), *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (pp. 490–503). ACM. https://doi.org/10.1145/3230543.3230582 (see p. 33).

Liu, M., Cui, T., Schuh, H., Krishnamurthy, A., Peter, S., & Gupta, K. (2019). Offloading distributed applications onto smartNICs using iPipe. In J. Wu & W. Hall (Eds.), *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (pp. 318–333). ACM. https://doi.org/10.1145/3341302.3342079 (see p. 46).

Liu, Z., Namkung, H., Nikolaidis, G., Lee, J., Kim, C., Jin, X., Braverman, V., Yu, M., & Sekar, V. (2021). Jaqen: A High-Performance Switch-Native Approach for Detecting and Mitigating Volumetric DDoS Attacks with Programmable Switches. In M. Bailey & R. Greenstadt (Eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (pp. 3829–3846). USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/liu-zaoxing (see p. 51).

Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., & Luo, J. (2007). NetFPGA-An Open Platform for Gigabit-Rate Network Switching and Routing. *IEEE International Conference on Microelectronic Systems Education, MSE '07, San Diego, CA, USA, June 3-4, 2007*, 160–161. https://doi.org/10.1109/MSE.2007.69 (see p. 26).

Lotfollahi, M., Siavoshani, M. J., Zade, R. S. H., & Saberian, M. (2020). Deep packet: a novel approach for encrypted traffic classification using deep learning. *Soft Comput.*, *24*(3), 1999–2012. https://doi.org/10.1007/s00500-019-04030-2 (see p. 64).

Luce, R. D. (1959). *Individual choice behavior.* John Wiley. (See p. 89).

Lyon, J. (2020, May 29). *AWS Shield Threat Landscape Report – Q1 2020.* Retrieved April 2, 2022, from https://aws-shield-tlr.s3.amazonaws.com/2020-Q1_AWS_Shield_TLR.pdf (see p. 111).

Maas, A. L., Hannun, A. Y., & Ng, A. Y. (2013, June 16). Rectifier Nonlinearities Improve Neural Network Acoustic Models. In B. Kingsbury, T. N. Sainath, L. Deng & A. Senior (Eds.), *Proceedings of the ICML 2013 workshop on Deep Learning for Audio, Speech and Language Processing*. JMLR.org. Retrieved November 1, 2021, from https://sites.google.com/site/deeplearningicml2013/relu_hybrid_icml2013_final.pdf (see p. 81).

Mach, S., Schuiki, F., Zaruba, F., & Benini, L. (2020). FPnew: An Open-Source Multi-Format Floating-Point Unit Architecture for Energy-

Proportional Transprecision Computing. *CoRR, abs/2007.01530*. https://arxiv.org/abs/2007.01530 (see p. 92).

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2018). Towards Deep Learning Models Resistant to Adversarial Attacks. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. https://openreview.net/forum?id=rJzIBfZAb (see p. 101).

Mah, B. A., Cziva, R., & Kumar, Y. (2020, September 25). *ESnet6 High Touch Services: Precision Streaming Network Telemetry* [Presented at the International Conference for High Performance Computing, Networking, Storage, and Analysis 2020 (SC '20).]. Retrieved March 10, 2022, from https://sc20.supercomputing.org/app/uploads/2020/12/06_sc20_xnet_mah_cziva_kumar_esnet6.pdf (see p. 29).

Mahajan, R., Bellovin, S. M., Floyd, S., Ioannidis, J., Paxson, V., & Shenker, S. (2002). Controlling high bandwidth aggregates in the network. *Computer Communication Review, 32*(3), 62–73. https://doi.org/10.1145/571697.571724 (see pp. 68, 114, 124).

Maiorca, D., Corona, I., & Giacinto, G. (2013). Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious PDF files detection. In K. Chen, Q. Xie, W. Qiu, N. Li & W. Tzeng (Eds.), *8th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '13, Hangzhou, China - May 08 - 10, 2013* (pp. 119–130). ACM. https://doi.org/10.1145/2484313.2484327 (see p. 101).

Malialis, K., & Kudenko, D. (2013). Multiagent Router Throttling: Decentralized Coordinated Response Against DDoS Attacks. In H. Muñoz-Avila & D. J. Stracuzzi (Eds.), *Proceedings of the Twenty-Fifth Innovative Applications of Artificial Intelligence Conference, IAAI 2013, July 14-18, 2013, Bellevue, Washington, USA*. AAAI. http://www.aaai.org/ocs/index.php/IAAI/IAAI13/paper/view/6244 (see pp. 67, 116).

Malialis, K., & Kudenko, D. (2015). Distributed response to network intrusions using multiagent reinforcement learning. *Eng. Appl. of AI, 41*, 270–284. https://doi.org/10.1016/j.engappai.2015.01.013 (see pp. 67, 110, 116, 118, 121, 135, 136, 215).

Mangla, T., Halepovic, E., Zegura, E. W., & Ammar, M. H. (2020). Drop the packets: using coarse-grained data to detect video performance issues. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 71–77). ACM. https://doi.org/10.1145/3386367.3431294 (see p. 71).

Mao, H., Alizadeh, M., Menache, I., & Kandula, S. (2016). Resource Management with Deep Reinforcement Learning. In B. Ford, A. C. Snoeren & E. W. Zegura (Eds.), *Proceedings of the 15th ACM Workshop on Hot Topics in Networks, HotNets 2016, Atlanta, GA, USA, November 9-10, 2016* (pp. 50–56). ACM. https://doi.org/10.1145/3005745.3005750 (see p. 71).

Mao, H., Netravali, R., & Alizadeh, M. (2017). Neural Adaptive Video Streaming with Pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017* (pp. 197–210). ACM. https://doi.org/10.1145/3098822.3098843 (see p. 70).

Mao, H., Schwarzkopf, M., Venkatakrishnan, S. B., Meng, Z., & Alizadeh, M. (2019). Learning scheduling algorithms for data processing clusters. In J. Wu & W. Hall (Eds.), *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (pp. 270–288). ACM. https://doi.org/10.1145/3341302.3342080 (see p. 71).

Martins, J., Ahmed, M., Raiciu, C., Olteanu, V. A., Honda, M., Bifulco, R., & Huici, F. (2014). ClickOS and the Art of Network Function Virtualization. In R. Mahajan & I. Stoica (Eds.), *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014* (pp. 459–473). USENIX Association. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/martins (see pp. 24, 183).

Mathis, M., Semke, J., Mahdavi, J., & Ott, T. (1997). The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communication Review, 27*(3), 67–82. https://doi.org/10.1145/263932.264023 (see pp. 68, 114).

McCanne, S., & Jacobson, V. (1993). The BSD Packet Filter: A New Architecture for User-level Packet Capture. *Proceedings of the Usenix Winter 1993 Technical Conference, San Diego, California, USA, January 1993*, 259–270. https://www.usenix.org/conference/usenix-winter-1993-conference/bsd-packet-filter-new-architecture-user-level-packet (see p. 42).

McCarthy, K. (2019, October 28). *Amazon is saying nothing about the DDoS attack that took down AWS, but others are.* Retrieved April 2, 2022, from https://www.theregister.com/2019/10/28/amazon_ddos_attack/ (see p. 111).

McCreesh, C. (2017). *Solving hard subgraph problems in parallel* [Doctoral dissertation, University of Glasgow, UK]. Retrieved April 13, 2022, from https://theses.gla.ac.uk/8322/ (see p. 211).

McKeown, N., Anderson, T. E., Balakrishnan, H., Parulkar, G. M., Peterson, L. L., Rexford, J., Shenker, S., & Turner, J. S. (2008). OpenFlow: enabling innovation in campus networks. *Comput. Commun. Rev., 38*(2), 69–74. https://doi.org/10.1145/1355734.1355746 (see p. 19).

McKeown, N., Talayco, D., Varghese, G., Lopes, N., Bjørner, N., & Rybalchenko, A. (2016). *Automatically verifying reachability and well-formedness in P4 Networks* (tech. rep. No. MSR-TR-2016-65). Retrieved March 11, 2022, from https://www.microsoft.com/en-us/research/publication/automatically-verifying-reachability-well-formedness-p4-networks/ (see p. 33).

McQuistin, S., Karan, M., Khare, P., Perkins, C., Tyson, G., Purver, M., Healey, P., Iqbal, W., Qadir, J., & Castro, I. (2021). Characterising the IETF through the lens of RFC deployment. In D. Levin, A. Mislove, J. Amann & M. Luckie (Eds.), *IMC '21: ACM Internet Measurement Conference, Virtual Event, USA, November 2-4, 2021* (pp. 137–149). ACM. https://doi.org/10.1145/3487552.3487821 (see p. 13).

Melis, L., Song, C., Cristofaro, E. D., & Shmatikov, V. (2019). Exploiting Unintended Feature Leakage in Collaborative Learning. *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, 691–706. https://doi.org/10.1109/SP.2019.00029 (see p. 105).

Meng, Z., Wang, M., Bai, J., Xu, M., Mao, H., & Hu, H. (2020). Interpreting Deep Learning-Based Networking Systems. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 154–171). ACM. https://doi.org/10.1145/3387514.3405859 (see p. 97).

Menon, A. (2021, April 27). *How will Intel's "Ice Lake" redefine the scope of data security?* Retrieved November 19, 2021, from https://fortanix.com/blog/2021/04/how-will-intels-ice-lake-redefine-the-scope-of-data-security/ (see p. 105).

Mestres, A., Rodríguez-Natal, A., Carner, J., Barlet-Ros, P., Alarcón, E., Solé, M., Muntés, V., Meyer, D., Barkai, S., Hibbett, M. J., Estrada, G., Maruf, K., Coras, F., Ermagan, V., Latapie, H., Cassar, C., Evans, J., Maino, F., Walrand, J. C., & Cabellos, A. (2016). Knowledge-Defined Networking. *CoRR, abs/1606.06222*. http://arxiv.org/abs/1606.06222 (see p. 60).

Meyer, K. (2020, January 17). *The RARE project, bringing back the network innovation within research and education community*. Retrieved March 10, 2022, from https://connect.geant.org/2020/01/17/rare-project-bringing-back-the-network-innovation-within-research-and-education-community (see p. 29).

Miano, S., Sanaee, A., Risso, F., Rétvári, G., & Antichi, G. (2022). Domain specific run time optimization for software data planes. In B. Falsafi, M. Ferdman, S. Lu & T. F. Wenisch (Eds.), *ASPLOS '22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022* (pp. 1148–1164). ACM. https://doi.org/10.1145/3503222.3507769 (see p. 48).

Migacz, S. (2017). *8-bit Inference with TensorRT*. Retrieved February 1, 2021, from https://on-demand.gputechconf.com/gtc/2017/presentation/s7310-8-bit-inference-with-tensorrt.pdf (see p. 93).

Mininet Project. (2022). *Mininet: An Instant Virtual Network on your Laptop (or other PC)*. Retrieved April 21, 2022, from http://mininet.org (see p. 133).

Mirhoseini, A., Goldie, A., Yazgan, M., Jiang, J. W., Songhori, E., Wang, S., Lee, Y.-J., Johnson, E., Pathak, O., Nazi, A., Pak, J., Tong, A., Srinivasa, K., Hang, W., Tuncer, E., Le, Q. V., Laudon, J., Ho, R., Carpenter, R., & Dean, J. (2021). A graph placement methodology for fast chip design. *Nature*, *594*(7862), 207–212. https://doi.org/10.1038/s41586-021-03544-w (see pp. 73, 82).

MIT Lincoln Labs. (2018). *DARPA Intrusion Detection Data Sets*. Retrieved August 7, 2018, from https://www.ll.mit.edu/r-d/datasets (see p. 67).

Miyashita, D., Lee, E. H., & Murmann, B. (2016). Convolutional Neural Networks using Logarithmic Data Representation. *CoRR*, *abs/1603.01025*. http://arxiv.org/abs/1603.01025 (see p. 93).

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T. P., Harley, T., Silver, D., & Kavukcuoglu, K. (2016). Asynchronous Methods for Deep Reinforcement Learning. In M. Balcan & K. Q. Weinberger (Eds.), *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016* (pp. 1928–1937, Vol. 48). JMLR.org. http://proceedings.mlr.press/v48/mniha16.html (see p. 88).

Molnar, C. (2021, October 19). *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Retrieved October 28, 2021, from https://christophm.github.io/interpretable-ml-book/ (see p. 96).

Molnar, C., Casalicchio, G., & Bischl, B. (2020). Interpretable Machine Learning - A Brief History, State-of-the-Art and Challenges. In I. Koprinska, M. Kamp, A. Appice, C. Loglisci, L. Antonie, A. Zimmermann, R. Guidotti, Ö. Özgöbek, R. P. Ribeiro, R. Gavaldà, J. Gama, L. Adilova, Y. Krishnamurthy, P. M. Ferreira, D. Malerba, I. Medeiros, M. Ceci, G. Manco, E. Masciari, … J. A. Gulla (Eds.), *ECML PKDD 2020 Workshops - Workshops of the European Conference on Machine Learning and Knowledge Discovery in Databases (ECML PKDD 2020): SoGood 2020, PDFL 2020, MLCS 2020, NFMCP 2020, DINA 2020, EDML 2020, XKDD 2020 and INRA 2020, Ghent, Belgium, September 14-18, 2020, Proceedings* (pp. 417–431, Vol. 1323). Springer. https://doi.org/10.1007/978-3-030-65965-3_28 (see p. 96).

Moore, A. W., & Zuev, D. (2005). Internet traffic classification using bayesian analysis techniques. In D. L. Eager, C. L. Williamson, S. C. Borst & J. C. S. Lui (Eds.), *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2005, June 6-10, 2005, Banff, Alberta, Canada* (pp. 50–60). ACM. https://doi.org/10.1145/1064212.1064220 (see p. 64).

Moore, D. (2003). Network Telescopes: Tracking Denial-of-Service Attacks and Internet Worms Around the Globe. In Æ. Frisch (Ed.), *Proceedings of the 17th Conference on Systems Administration (LISA 2003), San Diego, California, USA, October 26-31, 2003*. USENIX. Retrieved

April 2, 2022, from http://www.cs.unc.edu/~jeffay/courses/nidsS05/measurement/moore-telescopes04.pdf (see p. 112).

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics*, *38*(8). Retrieved February 4, 2022, from https://newsroom.intel.com/wp-content/uploads/sites/11/2018/05/moores-law-electronics.pdf (see p. 29).

Morris, R. T., Kohler, E., Jannotti, J., & Kaashoek, M. F. (1999). The Click modular router. In D. Kotz & J. Wilkes (Eds.), *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999* (pp. 217–231). ACM. https://doi.org/10.1145/319151.319166 (see pp. 15, 24).

Moura, G. C. M., Castro, S., Heidemann, J., & Hardaker, W. (2021). *TsuNAME vulnerability and DDoS against DNS* (tech. rep. No. ISI-TR-740). USC/Information Sciences Institute. https://www.isi.edu/%7ejohnh/PAPERS/Moura21a.html (see p. 114).

Nair, V., & Hinton, G. E. (2010). Rectified Linear Units Improve Restricted Boltzmann Machines. In J. Fürnkranz & T. Joachims (Eds.), *Proceedings of the 27th International Conference on Machine Learning (ICML-10), June 21-24, 2010, Haifa, Israel* (pp. 807–814). Omnipress. https://icml.cc/Conferences/2010/papers/432.pdf (see p. 81).

Netronome. (2021). *SmartNIC Overview*. Retrieved February 1, 2021, from https://www.netronome.com/products/smartnic/overview/ (see p. 26).

Neugebauer, R., Antichi, G., Zazo, J. F., Audzevich, Y., López-Buedo, S., & Moore, A. W. (2018). Understanding PCIe performance for end host networking. In S. Gorinsky & J. Tapolcai (Eds.), *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2018, Budapest, Hungary, August 20-25, 2018* (pp. 327–341). ACM. https://doi.org/10.1145/3230543.3230560 (see pp. 40, 147, 183).

Nguyen, X. N., Saucez, D., Barakat, C., & Turletti, T. (2016). Rules Placement Problem in OpenFlow Networks: A Survey. *IEEE Communications Surveys and Tutorials*, *18*(2), 1273–1286. https://doi.org/10.1109/COMST.2015.2506984 (see p. 144).

Nir, Y., & Langley, A. (2018). ChaCha20 and Poly1305 for IETF Protocols. https://doi.org/10.17487/RFC8439 (see p. 224).

Nokia. (2021). *Nokia launches fifth generation routing silicon, sets new benchmarks for IP network security and energy efficiency*. Retrieved September 27, 2021, from https://www.nokia.com/about-us/news/releases/2021/09/21/nokia-launches-fifth-generation-routing-silicon-sets-new-benchmarks-for-ip-network-security-and-energy-efficiency/ (see pp. 2, 31).

NPL. (2019). *NPL – Open, High-Level language for developing feature-rich solutions for programmable networking platforms*. Retrieved March 11, 2022, from https://nplang.org/ (see p. 32).

Nunes, B. A. A., Mendonca, M., Nguyen, X. N., Obraczka, K., & Turletti, T. (2014). A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE Commun. Surv. Tutorials*, *16*(3), 1617–1634. https://doi.org/10.1109/SURV.2014.012214.00180 (see p. 18).

NVIDIA. (2021a). *GPUDirect*. Retrieved October 14, 2021, from https://developer.nvidia.com/gpudirect (see p. 148).

NVIDIA. (2021b). *NVIDIA BlueField Data Processing Units*. Retrieved May 11, 2021, from https://www.nvidia.com/en-gb/networking/products/data-processing-unit/ (see p. 26).

Nygren, E., Garland, S., & Kaashoek, M. (1999). PAN: a high-performance active network node supporting multiple mobile code systems. *1999 IEEE Second Conference on Open Architectures and Network Programming. Proceedings. OPENARCH '99 (Cat. No.99EX252)*, 78–89. https://doi.org/10.1109/OPNARC.1999.758497 (see p. 15).

Olah, C., Mordvintsev, A., & Schubert, L. (2017, November 7). *Feature Visualisation*. https://doi.org/10.23915/distill.00007 (see p. 96).

Open Networking Foundation. (2015, March 26). *OpenFlow Switch Specification: Version 1.5.1 (Protocol version 0x06)*. Retrieved February 17, 2022, from https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf (see p. 20).

Open Networking Foundation. (2021, December 9). *Open Network Operating System (ONOS) SDN Controller for SDN/NFV Solutions*. Retrieved February 11, 2022, from https://opennetworking.org/onos/ (see p. 21).

Open vSwitch. (2018). Retrieved May 2, 2018, from https://www.openvswitch.org/ (see p. 119).

OpenDaylight Project. (2021). *OpenDaylight*. Retrieved February 11, 2022, from https://www.opendaylight.org/ (see p. 21).

P4c Maintainers. (2017). *P4c: eBPF Backend*. Retrieved March 21, 2022, from https://github.com/p4lang/p4c/blob/main/backends/ebpf/README.md (see p. 28).

Pagh, R., & Rodler, F. F. (2001). Cuckoo Hashing. In F. M. auf der Heide (Ed.), *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings* (pp. 121–133, Vol. 2161). Springer. https://doi.org/10.1007/3-540-44676-1_10 (see p. 194).

Pan, R., Breslau, L., Prabhakar, B., & Shenker, S. (2003). Approximate fairness through differential dropping. *Computer Communication Review*, *33*(2), 23–39. https://doi.org/10.1145/956981.956985 (see pp. 62, 115, 144).

Pan, R., Natarajan, P., Baker, F., & White, G. (2017). Proportional Integral Controller Enhanced (PIE): A Lightweight Control Scheme to Address the Bufferbloat Problem. https://doi.org/10.17487/RFC8033 (see p. 53).

Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., & Shenker, S. (2016). NetBricks: Taking the V out of NFV. In K. Keeton & T. Roscoe (Eds.), *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016* (pp. 203–216). USENIX Association. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda (see p. 24).

Pantos, R., & May, W. (2017). HTTP Live Streaming. https://doi.org/10.17487/RFC8216 (see p. 70).

Papernot, N. (2018). *Characterizing the Limits and Defenses of Machine Learning in Adversarial Settings* [Doctoral dissertation, Pennsylvania State University]. Retrieved June 28, 2021, from https://etda.libraries.psu.edu/catalog/15065ngp5056 (see p. 60).

Papernot, N., McDaniel, P. D., Jha, S., Fredrikson, M., Celik, Z. B., & Swami, A. (2016). The Limitations of Deep Learning in Adversarial Settings. In *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016* (pp. 372–387). IEEE. https://doi.org/10.1109/EuroSP.2016.36 (see pp. 98, 118).

Papernot, N., McDaniel, P. D., Sinha, A., & Wellman, M. P. (2018). SoK: Security and Privacy in Machine Learning. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018* (pp. 399–414). IEEE. https://doi.org/10.1109/EuroSP.2018.00035 (see pp. 98, 118).

Papernot, N., McDaniel, P. D., Wu, X., Jha, S., & Swami, A. (2016). Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016* (pp. 582–597). IEEE Computer Society. https://doi.org/10.1109/SP.2016.41 (see pp. 99, 100).

Park, W. H., & Ahn, S. (2017). Performance Comparison and Detection Analysis in Snort and Suricata Environment. *Wirel. Pers. Commun.*, *94*(2), 241–252. https://doi.org/10.1007/s11277-016-3209-9 (see p. 191).

Paxson, V. (1998). Bro: A System for Detecting Network Intruders in Real-Time. In A. D. Rubin (Ed.), *Proceedings of the 7th USENIX Security Symposium, San Antonio, TX, USA, January 26-29, 1998*. USENIX Association. https://www.usenix.org/conference/7th-usenix-security-symposium/bro-system-detecting-network-intruders-real-time (see pp. 24, 64).

Perkins, C., & Westerlund, M. (2010). Multiplexing RTP Data and Control Packets on a Single Port. https://doi.org/10.17487/RFC5761 (see p. 224).

Peterson, L. (2020, March 19). *It's Been a Fun Ride*. Retrieved February 7, 2022, from https://www.systemsapproach.org/blog-archive/its-been-a-fun-ride (see p. 14).

Pfaff, B., Pettit, J., Koponen, T., Jackson, E. J., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., Amidon, K., & Cas-

ADO, M. (2015). The Design and Implementation of Open vSwitch. *12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 15, Oakland, CA, USA, May 4-6, 2015*, 117–130. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/pfaff (see pp. 21, 119).

PHOTHILIMTHANA, P. M., LIU, M., KAUFMANN, A., PETER, S., BODÍK, R., & ANDERSON, T. E. (2018). Floem: A Programming System for NIC-Accelerated Network Applications. In A. C. ARPACI-DUSSEAU & G. VOELKER (Eds.), *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018* (pp. 663–679). USENIX Association. https://www.usenix.org/conference/osdi18/presentation/phothilimthana (see p. 46).

PIERAZZI, F., PENDLEBURY, F., CORTELLAZZI, J., & CAVALLARO, L. (2020). Intriguing Properties of Adversarial ML Attacks in the Problem Space. *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, 1332–1349. https://doi.org/10.1109/SP40000.2020.00073 (see p. 100).

PINHO, M. (2021, May 20). *AWS Shield threat landscape review: 2020 year-in-review*. Retrieved April 2, 2022, from https://aws.amazon.com/blogs/security/aws-shield-threat-landscape-review-2020-year-in-review/ (see p. 111).

PODDAR, R., LAN, C., POPA, R. A., & RATNASAMY, S. (2018). SafeBricks: Shielding Network Functions in the Cloud. In S. BANERJEE & S. SESHAN (Eds.), *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018* (pp. 201–216). USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/poddar (see p. 25).

PONTARELLI, S., BIFULCO, R., BONOLA, M., CASCONE, C., SPAZIANI, M., BRUSCHI, V., SANVITO, D., SIRACUSANO, G., CAPONE, A., HONDA, M., & HUICI, F. (2019). FlowBlaze: Stateful Packet Processing in Hardware. In J. R. LORCH & M. YU (Eds.), *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2019, Boston, MA, February 26-28, 2019* (pp. 531–548). USENIX Association. https://www.usenix.org/conference/nsdi19/presentation/pontarelli (see p. 34).

PRIMORAC, M., BUGNION, E., & ARGYRAKI, K. J. (2017). How to Measure the Killer Microsecond. *Comput. Commun. Rev.*, *47*(5), 61–66. https://doi.org/10.1145/3155055.3155065 (see p. 199).

QIN, Q., POULARAKIS, K., LEUNG, K. K., & TASSIULAS, L. (2020). Line-Speed and Scalable Intrusion Detection at the Network Edge via Federated Learning. *2020 IFIP Networking Conference, Networking 2020, Paris, France, June 22-26, 2020*, 352–360. https://ieeexplore.ieee.org/document/9142704 (see p. 68).

RAGHUNATHAN, A., STEINHARDT, J., & LIANG, P. (2018). Certified Defenses against Adversarial Examples. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 -*

May 3, 2018, Conference Track Proceedings. https://openreview.net/forum?id=Bys4ob-Rb (see p. 101).

Ramanathan, S., Mirkovic, J., Yu, M., & Zhang, Y. (2018). SENSS Against Volumetric DDoS Attacks. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018* (pp. 266–277). ACM. https://doi.org/10.1145/3274694.3274717 (see pp. 117, 144).

Rashelbach, A., Rottenstreich, O., & Silberstein, M. (2020). A Computational Approach to Packet Classification. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 542–556). ACM. https://doi.org/10.1145/3387514.3405886 (see p. 63).

Rastegari, M., Ordonez, V., Redmon, J., & Farhadi, A. (2016). XNOR-Net: ImageNet Classification Using Binary Convolutional Neural Networks. In B. Leibe, J. Matas, N. Sebe & M. Welling (Eds.), *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV* (pp. 525–542, Vol. 9908). Springer. https://doi.org/10.1007/978-3-319-46493-0_32 (see p. 93).

Reddy, T., Johnston, A., Matthews, P., & Rosenberg, J. (2020). Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). https://doi.org/10.17487/RFC8656 (see p. 223).

Rhee, I., Xu, L., Ha, S., Zimmermann, A., Eggert, L., & Scheffenegger, R. (2018). CUBIC for Fast Long-Distance Networks. https://doi.org/10.17487/RFC8312 (see pp. 2, 68, 114, 134).

Ribeiro, M. T., Singh, S., & Guestrin, C. (2016). "Why Should I Trust You?": Explaining the Predictions of Any Classifier. In B. Krishnapuram, M. Shah, A. J. Smola, C. C. Aggarwal, D. Shen & R. Rastogi (Eds.), *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016* (pp. 1135–1144). ACM. https://doi.org/10.1145/2939672.2939778 (see p. 96).

Richter, P., & Berger, A. W. (2019). Scanning the Scanners: Sensing the Internet from a Massively Distributed Network Telescope. *Proceedings of the Internet Measurement Conference, IMC 2019, Amsterdam, The Netherlands, October 21-23, 2019*, 144–157. https://doi.org/10.1145/3355369.3355595 (see p. 112).

Riera, J. F., Batalle, J., Bonnet, J., Dias, M., McGrath, M. J., Petralia, G., Liberati, F., Giuseppi, A., Pietrabissa, A., Ceselli, A., Petrini, A., Trubian, M., Papadimitriou, P., Dietrich, D., Ramos, A., Melian, J., Xilouris, G., Kourtis, A., Kourtis, T., & Markakis, E. K. (2016). TeNOR: Steps towards an orchestration platform for multi-PoP NFV deployment. *IEEE NetSoft Conference and Workshops, NetSoft 2016,*

Seoul, South Korea, June 6-10, 2016, 243–250. https://doi.org/10.1109/NETSOFT.2016.7502419 (see p. 24).

Rizzo, L. (2012). netmap: A Novel Framework for Fast Packet I/O. In G. Heiser & W. C. Hsieh (Eds.), *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012* (pp. 101–112). USENIX Association. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/rizzo (see p. 42).

Roesch, M. (1999). Snort: Lightweight Intrusion Detection for Networks. In D. W. Parter (Ed.), *Proceedings of the 13th Conference on Systems Administration (LISA-99), Seattle, WA, USA, November 7-12, 1999* (pp. 229–238). USENIX. http://www.usenix.org/publications/library/proceedings/lisa99/roesch.html (see pp. 24, 64, 191).

Rossi, D., Testa, C., Valenti, S., & Muscariello, L. (2010). LEDBAT: The New BitTorrent Congestion Control Protocol. *Proceedings of the 19th International Conference on Computer Communications and Networks, IEEE ICCCN 2010, Zürich, Switzerland, August 2-5, 2010*, 1–6. https://doi.org/10.1109/ICCCN.2010.5560080 (see p. 114).

Rossow, C. (2014). Amplification Hell: Revisiting Network Protocols for DDoS Abuse. In *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society. https://www.ndss-symposium.org/ndss2014/amplification-hell-revisiting-network-protocols-ddos-abuse (see pp. 113, 116, 118, 135).

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, *323*(6088), 533–536. https://doi.org/10.1038/323533a0 (see p. 82).

Rüth, J., Poese, I., Dietzel, C., & Hohlfeld, O. (2018). A First Look at QUIC in the Wild. In R. Beverly, G. Smaragdakis & A. Feldmann (Eds.), *Passive and Active Measurement - 19th International Conference, PAM 2018, Berlin, Germany, March 26-27, 2018, Proceedings* (pp. 255–268, Vol. 10771). Springer. https://doi.org/10.1007/978-3-319-76481-8_19 (see pp. 124, 217).

Ryu SDN Framework Community. (2017). *Ryu SDN Framework*. Retrieved February 11, 2022, from https://ryu-sdn.org/ (see pp. 21, 133).

Rzadca, K., Findeisen, P., Swiderski, J., Zych, P., Broniek, P., Kusmierek, J., Nowak, P., Strack, B., Witusowski, P., Hand, S., & Wilkes, J. (2020). Autopilot: workload autoscaling at Google. In A. Bilas, K. Magoutis, E. P. Markatos, D. Kostic & M. I. Seltzer (Eds.), *EuroSys '20: Fifteenth EuroSys Conference 2020, Heraklion, Greece, April 27-30, 2020* (16:1–16:16). ACM. https://doi.org/10.1145/3342195.3387524 (see p. 72).

Salimans, T., Ho, J., Chen, X., & Sutskever, I. (2017). Evolution Strategies as a Scalable Alternative to Reinforcement Learning. *CoRR, abs/1703.03864*. http://arxiv.org/abs/1703.03864 (see p. 88).

Saltzer, J. H., Reed, D. P., & Clark, D. D. (1984). End-To-End Arguments in System Design. *ACM Trans. Comput. Syst.*, *2*(4), 277–288. https://doi.org/10.1145/357401.357402 (see p. 11).

Sander, C., Rüth, J., Hohlfeld, O., & Wehrle, K. (2019). DeePCCI: Deep Learning-based Passive Congestion Control Identification. *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*, 37–43. https://doi.org/10.1145/3341216.3342211 (see p. 64).

Sanvito, D., Siracusano, G., & Bifulco, R. (2018). Can the Network be the AI Accelerator? In X. Jin & C. Kim (Eds.), *Proceedings of the 2018 Morning Workshop on In-Network Computing, NetCompute@SIGCOMM 2018, Budapest, Hungary, August 20, 2018* (pp. 20–25). ACM. https://doi.org/10.1145/3229591.3229594 (see pp. 55, 93).

Sapio, A., Abdelaziz, I., Canini, M., & Kalnis, P. (2017). DAIET: a system for data aggregation inside the network. In *Proceedings of the 2017 Symposium on Cloud Computing, SoCC 2017, Santa Clara, CA, USA, September 24-27, 2017* (p. 626). ACM. https://doi.org/10.1145/3127479.3132018 (see p. 52).

Sapio, A., Canini, M., Ho, C., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D. R. K., & Richtárik, P. (2021). Scaling Distributed Machine Learning with In-Network Aggregation. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 785–808). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/sapio (see pp. 56, 93).

Schulzrinne, H., Casner, S. L., Frederick, R., & Jacobson, V. (2003). RTP: A Transport Protocol for Real-Time Applications. https://doi.org/10.17487/RFC3550 (see p. 224).

Schwartz, B., Jackson, A. W., Strayer, W. T., Zhou, W., Rockwell, R. D., & Partridge, C. (2000). Smart packets: applying active networks to network management. *ACM Trans. Comput. Syst.*, *18*(1), 67–88. https://doi.org/10.1145/332799.332893 (see pp. 15, 31).

Settles, B. (2010, January 26). *Active Learning Literature Survey* (Computer Sciences Technical Report). University of Wisconsin-Madison. Retrieved May 10, 2018, from http://burrsettles.com/pub/settles.activelearning.pdf (see pp. 98, 102).

Shah, N., & Keutzer, K. (2002). Network Processors: Origin of Species. In I. Cicekli, N. K. Cicekli & E. Gelenbe (Eds.), *Proceedings of ISCIS XVII, The Seventeenth International Symposium on Computer and Information Sciences*. CRC Press. Retrieved March 4, 2022, from https://www.researchgate.net/publication/228421036_Network_processors_Origin_of_species (see p. 25).

Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N., & Rexford, J. (2016). PISCES: A Programmable, Protocol-Independent Software Switch. In M. P. Barcellos, J. Crowcroft, A. Vahdat & S. Katti (Eds.), *Proceedings of the ACM SIGCOMM 2016 Conference,*

Florianopolis, Brazil, August 22-26, 2016 (pp. 525–538). ACM. https://doi.org/10.1145/2934872.2934886 (see p. 28).

Shahinfar, F., Miano, S., Sanaee, A., Siracusano, G., Bifulco, R., & Antichi, G. (2021). The case for network functions decomposition. In G. Carle & J. Ott (Eds.), *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021* (pp. 475–476). ACM. https://doi.org/10.1145/3485983.3493349 (see p. 47).

Shen, S., Tople, S., & Saxena, P. (2016). Auror: defending against poisoning attacks in collaborative deep learning systems. In S. Schwab, W. K. Robertson & D. Balzarotti (Eds.), *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016* (pp. 508–519). ACM. https://doi.org/10.1145/2991079 (see p. 103).

Sherry, J., Hasan, S., Scott, C., Krishnamurthy, A., Ratnasamy, S., & Sekar, V. (2012). Making middleboxes someone else's problem: network processing as a cloud service. In L. Eggert, J. Ott, V. N. Padmanabhan & G. Varghese (Eds.), *ACM SIGCOMM 2012 Conference, SIGCOMM '12, Helsinki, Finland - August 13 - 17, 2012* (pp. 13–24). ACM. https://doi.org/10.1145/2342356.2342359 (see p. 23).

Shirokov, N., & Dasineni, R. (2018, May 22). *Open-sourcing Katran, a scalable network load balancer.* Retrieved March 28, 2022, from https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/ (see p. 51).

Shokri, R., Stronati, M., Song, C., & Shmatikov, V. (2017). Membership Inference Attacks Against Machine Learning Models. *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 3–18. https://doi.org/10.1109/SP.2017.41 (see p. 105).

Shrivastav, V. (2019). Fast, scalable, and programmable packet scheduler in hardware. In J. Wu & W. Hall (Eds.), *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM 2019, Beijing, China, August 19-23, 2019* (pp. 367–379). ACM. https://doi.org/10.1145/3341302.3342090 (see p. 53).

Shukla, A., Hudemann, K. N., Hecker, A., & Schmid, S. (2019). Runtime Verification of P4 Switches with Reinforcement Learning. *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*, 1–7. https://doi.org/10.1145/3341216.3342206 (see p. 69).

Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., & Riedmiller, M. A. (2014). Deterministic Policy Gradient Algorithms. *Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, 32*, 387–395. http://proceedings.mlr.press/v32/silver14.html (see p. 88).

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T. P., Hui, F., Sifre, L., van den Driessche, G., Graepel, T.,

& Hassabis, D. (2017). Mastering the game of Go without human knowledge. *Nat.*, *550*(7676), 354–359. https://doi.org/10.1038/nature24270 (see p. 3).

Silver, D., Singh, S. P., Precup, D., & Sutton, R. S. (2021). Reward is enough. *Artif. Intell.*, *299*, 103535. https://doi.org/10.1016/j.artint.2021.103535 (see p. 210).

Simon, M., Stubbe, H., Scholz, D., Gallenmüller, S., & Carle, G. (2021). High-Performance Match-Action Table Updates from within Programmable Software Data Planes. *ANCS '21: Symposium on Architectures for Networking and Communications Systems, Layfette, IN, USA, December 13 - 16, 2021*, 102–108. https://doi.org/10.1145/3493425.3502759 (see p. 36).

Simpson, K. A., Cziva, R., Kumar, Y., & Guok, C. (2019). Real-time Performance Analysis of High-Speed, International Science Network Flows. In R. Holz (Ed.), *Internet Measurement Conference Posters, IMC 2019, Amsterdam, The Netherlands, October 21-23, 2019*. ACM. Retrieved May 9, 2022, from https://mcfelix.me/docs/talks/imc-hightouch-poster.pdf (see p. 201).

Simpson, K. A., Cziva, R., & Pezaros, D. P. (2020). Seiðr: Dataplane Assisted Flow Classification Using ML. *IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, Taiwan, December 7-11, 2020*, 1–6. https://doi.org/10.1109/GLOBECOM42002.2020.9348063 (see pp. v, 192).

Simpson, K. A., & Pezaros, D. P. (2021). Online RL in the programmable dataplane with OPaL. In G. Carle & J. Ott (Eds.), *CoNEXT '21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021* (pp. 471–472). ACM. https://doi.org/10.1145/3485983.3493345 (see pp. v, 148).

Simpson, K. A., & Pezaros, D. P. (2022). Revisiting the Classics: Online RL in the Programmable Dataplane [SICSA PhD Conference Best Paper '22]. *2022 IEEE/IFIP Network Operations and Management Symposium, NOMS 2022, Budapest, Hungary, April 25-29, 2022*, 1–10. https://doi.org/10.1109/NOMS54207.2022.9789930 (see pp. v, 148).

Simpson, K. A., Rogers, S., & Pezaros, D. P. (2020). Per-Host DDoS Mitigation by Direct-Control Reinforcement Learning. *IEEE Trans. Network and Service Management*, *17*(1), 103–117. https://doi.org/10.1109/TNSM.2019.2960202 (see pp. v, 90, 110).

Simpson, S., Shirazi, S. N., Marnerides, A. K., Jouët, S., Pezaros, D., & Hutchison, D. (2018). An Inter-Domain Collaboration Scheme to Remedy DDoS Attacks in Computer Networks. *IEEE Trans. Network and Service Management*, *15*(3), 879–893. https://doi.org/10.1109/TNSM.2018.2828938 (see p. 117).

Singer, D., Desineni, H., & Even, R. (2017). A General Mechanism for RTP Header Extensions. https://doi.org/10.17487/RFC8285 (see p. 226).

Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., Kanagala, A., Provost, J., Simmons, J., Tanda, E., Wanderer, J., Hölzle, U., Stuart, S., & Vahdat, A. (2015). Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In S. Uhlig, O. Maennel, B. Karp & J. Padhye (Eds.), *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015* (pp. 183–197). ACM. https://doi.org/10.1145/2785956.2787508 (see p. 21).

Siracusano, G., & Bifulco, R. (2018). In-network Neural Networks. *CoRR, abs/1801.05731*. http://arxiv.org/abs/1801.05731 (see pp. 55, 93).

Siracusano, G., Galea, S., Sanvito, D., Malekzadeh, M., Haddadi, H., Antichi, G., & Bifulco, R. (2020). Running Neural Networks on the NIC. *CoRR, abs/2009.02353*. https://arxiv.org/abs/2009.02353 (see pp. 55, 94, 147, 187, 213).

Sivakumar, V., Rocktäschel, T., Miller, A. H., Küttler, H., Nardelli, N., Rabbat, M., Pineau, J., & Riedel, S. (2019). MVFST-RL: An Asynchronous RL Framework for Congestion Control with Delayed Actions. *CoRR, abs/1910.04054*. http://arxiv.org/abs/1910.04054 (see pp. 65, 90, 184).

Sivaraman, A., Cheung, A., Budiu, M., Kim, C., Alizadeh, M., Balakrishnan, H., Varghese, G., McKeown, N., & Licking, S. (2016). Packet Transactions: High-Level Programming for Line-Rate Switches. In M. P. Barcellos, J. Crowcroft, A. Vahdat & S. Katti (Eds.), *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (pp. 15–28). ACM. https://doi.org/10.1145/2934872.2934900 (see p. 34).

Sivaraman, A., Subramanian, S., Alizadeh, M., Chole, S., Chuang, S., Agrawal, A., Balakrishnan, H., Edsall, T., Katti, S., & McKeown, N. (2016). Programmable Packet Scheduling at Line Rate. In M. P. Barcellos, J. Crowcroft, A. Vahdat & S. Katti (Eds.), *Proceedings of the ACM SIGCOMM 2016 Conference, Florianopolis, Brazil, August 22-26, 2016* (pp. 44–57). ACM. https://doi.org/10.1145/2934872.2934899 (see pp. 34, 53).

Smith, J. M., & Schuchard, M. (2018). Routing Around Congestion: Defeating DDoS Attacks and Adverse Network Conditions via Reactive BGP Routing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA* (pp. 599–617). IEEE. https://doi.org/10.1109/SP.2018.00032 (see pp. 114, 116).

Smutz, C., & Stavrou, A. (2016). When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society. http://wp.internetsociety.org/ndss/wp-content/uploads/sit

es/25/2017/09/when-tree-falls-using-diversity-ensemble-classifiers-identify-evasion-malware-detectors.pdf (see p. 101).

Snort Team. (2017). *SNORT® Users Manual* (2.9.11). Retrieved November 10, 2017, from https://www.snort.org/documents/snort-users-manual (see pp. 24, 64).

Sommer, R., & Paxson, V. (2010). Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, 305–316. https://doi.org/10.1109/SP.2010.25 (see p. 67).

Sonchack, J., Loehr, D., Rexford, J., & Walker, D. (2021). Lucid: a language for control in the data plane. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021* (pp. 731–747). ACM. https://doi.org/10.1145/3452296.3472903 (see p. 32).

Song, E., Pan, T., Jia, C., Cao, W., Zhang, J., Huang, T., & Liu, Y. (2021). INT-label: Lightweight In-band Network-Wide Telemetry via Interval-based Distributed Labelling. *40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021*, 1–10. https://doi.org/10.1109/INFOCOM42981.2021.9488799 (see p. 50).

Song, H. (2013). Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane. In N. Foster & R. Sherwood (Eds.), *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking, HotSDN 2013, The Chinese University of Hong Kong, Hong Kong, China, Friday, August 16, 2013* (pp. 127–132). ACM. https://doi.org/10.1145/2491185.2491190 (see p. 27).

Soni, H., Rifai, M., Kumar, P., Doenges, R., & Foster, N. (2020). Composing Dataplane Programs with μP4. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 329–343). ACM. https://doi.org/10.1145/3387514.3405872 (see p. 32).

Sperl, P., Kao, C., Chen, P., Lei, X., & Böttinger, K. (2020). DLA: Dense-Layer-Analysis for Adversarial Example Detection. *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, 198–215. https://doi.org/10.1109/EuroSP48549.2020.00021 (see p. 102).

Srndic, N., & Laskov, P. (2014). Practical Evasion of a Learning-Based Classifier: A Case Study. *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, 197–211. https://doi.org/10.1109/SP.2014.20 (see p. 101).

Stephens, B., Akella, A., & Swift, M. M. (2018). Your Programmable NIC Should be a Programmable Switch. In *Proceedings of the 17th ACM*

Workshop on Hot Topics in Networks, HotNets 2018, Redmond, WA, USA, November 15-16, 2018 (pp. 36–42). ACM. https://doi.org/10.1145/3286062.3286068 (see p. 33).

Stewart, R. R. (2007). Stream Control Transmission Protocol. https://doi.org/10.17487/RFC4960 (see pp. 22, 114).

Stoyanov, R., & Zilberman, N. (2020). MTPSA: Multi-Tenant Programmable Switches. *EuroP4@CoNEXT 2020: Proceedings of the 3rd P4 Workshop in Europe, Barcelona, Spain, December 1, 2020*, 43–48. https://doi.org/10.1145/3426744.3431329 (see p. 34).

Studer, A., & Perrig, A. (2009). The Coremelt Attack. In M. Backes & P. Ning (Eds.), *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings* (pp. 37–52, Vol. 5789). Springer. https://doi.org/10.1007/978-3-642-04444-1_3 (see p. 114).

Sultana, N., Sonchack, J., Giesen, H., Pedisich, I., Han, Z., Shyamkumar, N., Burad, S., DeHon, A., & Loo, B. T. (2021). Flightplan: Dataplane Disaggregation and Placement for P4 Programs. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 571–592). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/sultana (see p. 48).

Sun, X., Choi, J., Chen, C., Wang, N., Venkataramani, S., Srinivasan, V., Cui, X., Zhang, W., & Gopalakrishnan, K. (2019). Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada* (pp. 4901–4910). https://proceedings.neurips.cc/paper/2019/hash/65fc9fb4897a89789352e211ca2d398f-Abstract.html (see p. 92).

Sutherland, M., Gupta, S., Falsafi, B., Marathe, V. J., Pnevmatikatos, D. N., & Daglis, A. (2020). The NEBULA RPC-Optimized Architecture. *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020*, 199–212. https://doi.org/10.1109/ISCA45697.2020.00027 (see pp. 35, 39).

Sutter, H. (2005). The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, *30*(3). Retrieved May 11, 2022, from http://www.gotw.ca/publications/concurrency-ddj.htm (see p. 211).

Sutton, R. S., & Barto, A. G. (2018). *Reinforcement Learning: An Introduction* (F. Bach, Ed.; 2nd ed.). MIT Press. http://incompleteideas.net/book/the-book-2nd.html (see pp. 78, 84, 86, 120).

Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., & Olukotun, K. (2022). Taurus: a data plane architecture for per-packet ML. In B. Falsafi, M. Ferdman, S. Lu & T. F. Wenisch (Eds.), *ASPLOS '22: 27th ACM*

International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022 (pp. 1099–1114). ACM. https://doi.org/10.1145/3503222.3507726 (see pp. 35, 93, 192).

Swamy, T., Rucker, A., Shahbaz, M., & Olukotun, K. (2020). Taurus: An Intelligent Data Plane. *CoRR, abs/2002.08987*. https://arxiv.org/abs/2002.08987 (see pp. 35, 93, 187).

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I. J., & Fergus, R. (2013). Intriguing properties of neural networks. *CoRR, abs/1312.6199*. http://arxiv.org/abs/1312.6199 (see p. 99).

Takruri, H., Kettaneh, I., Alquraan, A., & Al-Kiswany, S. (2020). FLAIR: Accelerating Reads with Consistency-Aware Network Routing. In R. Bhagwan & G. Porter (Eds.), *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (pp. 723–737). USENIX Association. https://www.usenix.org/conference/nsdi20/presentation/takruri (see p. 51).

Tan, T. J. L., & Shokri, R. (2020). Bypassing Backdoor Detection Algorithms in Deep Learning. *IEEE European Symposium on Security and Privacy, EuroS&P 2020, Genoa, Italy, September 7-11, 2020*, 175–183. https://doi.org/10.1109/EuroSP48549.2020.00019 (see p. 103).

Tavallaee, M., Bagheri, E., Lu, W., & Ghorbani, A. A. (2009). A detailed analysis of the KDD CUP 99 data set. *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications, CISDA 2009, Ottawa, Canada, July 8-10, 2009*, 1–6. https://doi.org/10.1109/CISDA.2009.5356528 (see p. 67).

Taylor, D. E., & Turner, J. S. (2005). Scalable Packet Classification using Distributed Crossproducting of Field Labels. *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies, 13-17 March 2005, Miami, FL, USA*, 269–280. https://doi.org/10.1109/INFCOM.2005.1497898 (see pp. 36, 154).

Tennenhouse, D. L., & Wetherall, D. (1996). Towards an active network architecture. *Comput. Commun. Rev., 26*(2), 5–17. https://doi.org/10.1145/231699.231701 (see pp. 3, 14).

Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. *Commun. ACM, 38*(3), 58–68. https://doi.org/10.1145/203330.203343 (see p. 88).

Thalheim, J., Unnibhavi, H., Priebe, C., Bhatotia, P., & Pietzuch, P. R. (2021). rkt-io: a direct I/O stack for shielded execution. In A. Barbalace, P. Bhatotia, L. Alvisi & C. Cadar (Eds.), *EuroSys '21: Sixteenth European Conference on Computer Systems, Online Event, United Kingdom, April 26-28, 2021* (pp. 490–506). ACM. https://doi.org/10.1145/3447786.3456255 (see p. 42).

The Linux Foundation. (2020). *Myth-busting DPDK in 2020: Revealed: the past, present, and future of the most popular data plane development*

kit in the world. (tech. rep.). Retrieved March 17, 2022, from https://nextgeninfra.io/wp-content/uploads/2020/07/AvidThink-Linux-Foundation-Myth-busting-DPDK-in-2020-Research-Brief-REV-B.pdf (see p. 42).

The Moving Picture Experts Group. (2021). *MPEG-DASH: Dynamic Adaptive Streaming over HTTP*. Retrieved August 26, 2021, from https://mpeg.chiariglione.org/standards/mpeg-dash (see p. 70).

The P4 Language Consortium. (2021, May 18). *P4 Portable NIC Architecture (PNA): version 0.5* [Working Draft.]. Retrieved March 8, 2022, from https://p4.org/p4-spec/docs/PNA.html (see pp. 36, 155).

The P4.org API Working Group. (2021, July 2). *P4Runtime Specification: Version 1.3.0* [Working draft.]. Retrieved February 4, 2022, from https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.pdf (see p. 28).

The P4.org Applications Working Group. (2020, November 11). *In-band Network Telemetry (INT) Dataplane Specification: Version 2.1*. Retrieved February 4, 2022, from https://p4.org/p4-spec/docs/INT_v2_1.pdf (see pp. 29, 50).

The P4.org Architecture Working Group. (2021). *$P4_{16}$ Portable Switch Architecture (PSA)* [Working Draft.]. Retrieved February 1, 2021, from https://p4.org/p4-spec/docs/PSA.html (see pp. 28, 152).

The Rust Team. (2022). *Rust Programming Language*. Retrieved February 28, 2022, from https://www.rust-lang.org/ (see p. 24).

The Zeek Project. (2020). *The Zeek Network Security Monitor*. Retrieved February 23, 2022, from https://zeek.org/ (see pp. 24, 64).

Tian, B., Gao, J., Liu, M., Zhai, E., Chen, Y., Zhou, Y., Dai, L., Yan, F., Ma, M., Tang, M., Lu, J., Wei, X., Liu, H. H., Zhang, M., Tian, C., & Yu, M. (2021). Aquila: a practically usable verification system for production-scale programmable data planes. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021* (pp. 17–32). ACM. https://doi.org/10.1145/3452296.3472937 (see pp. 28, 29, 33).

Tieleman, T., & Hinton, G. (2012). *Neural Networks for Machine Learning Lecture 6e: rmsprop: Divide the gradient by a running average of its recent magnitude*. Retrieved September 24, 2021, from https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf (see p. 84).

Tokic, M. (2010). Adaptive epsilon-Greedy Exploration in Reinforcement Learning Based on Value Difference. In R. Dillmann, J. Beyerer, U. D. Hanebeck & T. Schultz (Eds.), *KI 2010: Advances in Artificial Intelligence, 33rd Annual German Conference on AI, Karlsruhe, Germany, September 21-24, 2010. Proceedings* (pp. 203–210, Vol. 6359). Springer. https://doi.org/10.1007/978-3-642-16111-7_23 (see p. 89).

Tokic, M., & Palm, G. (2011). Value-Difference Based Exploration: Adaptive Control between Epsilon-Greedy and Softmax. In J. Bach & S. Edelkamp (Eds.), *KI 2011: Advances in Artificial Intelligence, 34th*

Annual German Conference on AI, Berlin, Germany, October 4-7,2011. Proceedings (pp. 335–346, Vol. 7006). Springer. https://doi.org/10.1007/978-3-642-24455-1_33 (see p. 89).

Tokic, M., & Palm, G. (2012). Gradient Algorithms for Exploration/Exploitation Trade-Offs: Global and Local Variants. In N. Mana, F. Schwenker & E. Trentin (Eds.), *Artificial Neural Networks in Pattern Recognition - 5th INNS IAPR TC 3 GIRPR Workshop, ANNPR 2012, Trento, Italy, September 17-19, 2012. Proceedings* (pp. 60–71, Vol. 7477). Springer. https://doi.org/10.1007/978-3-642-33212-8_6 (see p. 89).

Tokusashi, Y. (2021, July 1). *NetFPGA-PLUS: the next generation of NetFPGA platforms* [Presented at the 33rd Multi-Service Networks workshop (MSN'21)]. Retrieved March 4, 2022, from https://coseners.net/wp-content/uploads/2021/07/yuta_tokusashi_2021.pdf (see p. 26).

Tramèr, F., Behrmann, J., Carlini, N., Papernot, N., & Jacobsen, J. (2020). Fundamental Tradeoffs between Invariance and Sensitivity to Adversarial Perturbations. *CoRR, abs/2002.04599*. https://arxiv.org/abs/2002.04599 (see p. 102).

Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I. J., Boneh, D., & McDaniel, P. D. (2018). Ensemble Adversarial Training: Attacks and Defenses. *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. https://openreview.net/forum?id=rkZvSe-RZ (see p. 101).

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., & Ristenpart, T. (2016). Stealing Machine Learning Models via Prediction APIs. In T. Holz & S. Savage (Eds.), *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* (pp. 601–618). USENIX Association. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/tramer (see pp. 104–106, 118).

Travnik, J. B., Mathewson, K. W., Sutton, R. S., & Pilarski, P. M. (2018). Reactive Reinforcement Learning in Asynchronous Environments. *Front. Robotics and AI, 2018*. https://doi.org/10.3389/frobt.2018.00079 (see pp. 89, 90, 159).

Truong, J., Maini, P., Walls, R. J., & Papernot, N. (2021). Data-Free Model Extraction. *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, 4771–4780. https://openaccess.thecvf.com/content/CVPR2021/html/Truong_Data-Free_Model_Extraction_CVPR_2021_paper.html (see p. 105).

Tsilias, P. (2020, June 29). *What are the limits of Go channels, and just how fast are they?* Retrieved February 21, 2022, from https://tpaschalis.github.io/channels-limitations-speed/ (see p. 166).

Tu, W., Wei, Y., Antichi, G., & Pfaff, B. (2021). revisiting the open vSwitch dataplane ten years later. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27,*

2021 (pp. 245–257). ACM. https://doi.org/10.1145/3452296.3472914 (see pp. 21, 43).

Uhlenbeck, G. E., & Ornstein, L. S. (1930). On the Theory of the Brownian Motion. *Phys. Rev.*, *36*, 823–841. https://doi.org/10.1103/PhysRev.36.823 (see p. 89).

Valadarsky, A., Schapira, M., Shahaf, D., & Tamar, A. (2017). Learning to Route. In S. Banerjee, B. Karp & M. Walfish (Eds.), *Proceedings of the 16th ACM Workshop on Hot Topics in Networks, Palo Alto, CA, USA, HotNets 2017, November 30 - December 01, 2017* (pp. 185–191). ACM. https://doi.org/10.1145/3152434.3152441 (see pp. 61, 62).

van der Merwe, J. E., Rooney, S., Leslie, L., & Crosby, S. (1998). The Tempest – a practical framework for network programmability. *IEEE Netw.*, *12*(3), 20–28. https://doi.org/10.1109/65.690958 (see p. 18).

Vanaubel, Y., Mérindol, P., Pansiot, J., & Donnet, B. (2015). MPLS Under the Microscope: Revealing Actual Transit Path Diversity. In K. Cho, K. Fukuda, V. S. Pai & N. Spring (Eds.), *Proceedings of the 2015 ACM Internet Measurement Conference, IMC 2015, Tokyo, Japan, October 28-30, 2015* (pp. 49–62). ACM. https://doi.org/10.1145/2815675.2815687 (see p. 69).

Vasseur, J., Farrel, A., & Ash, G. (2006). A Path Computation Element (PCE)-Based Architecture. https://doi.org/10.17487/RFC4655 (see p. 19).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is All you Need. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan & R. Garnett (Eds.), *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA* (pp. 5998–6008). https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html

Vaucher, S., Yazdani, N., Felber, P., Lucani, D. E., & Schiavoni, V. (2020). ZipLine: in-network compression at line speed. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 399–405). ACM. https://doi.org/10.1145/3386367.3431302 (see p. 52).

Vörös, P., Horpácsi, D., Kitlei, R., Leskó, D., Tejfel, M., & Laki, S. (2018). T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors. *IEEE 19th International Conference on High Performance Switching and Routing, HPSR 2018, Bucharest, Romania, June 18-20, 2018*, 1–8. https://doi.org/10.1109/HPSR.2018.8850752 (see p. 28).

Wang, B., Yao, Y., Shan, S., Li, H., Viswanath, B., Zheng, H., & Zhao, B. Y. (2019). Neural Cleanse: Identifying and Mitigating Backdoor Attacks in Neural Networks. *2019 IEEE Symposium on Security and*

Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019, 707–723. https://doi.org/10.1109/SP.2019.00031 (see p. 103).

Wang, F., Wang, F., Liu, J., Shea, R., & Sun, L. (2020). Intelligent Video Caching at Network Edge: A Multi-Agent Deep Reinforcement Learning Approach. *39th IEEE Conference on Computer Communications, INFOCOM 2020, Toronto, ON, Canada, July 6-9, 2020*, 2499–2508. https://doi.org/10.1109/INFOCOM41043.2020.9155373 (see p. 73).

Wang, H., Liu, Z., & Shen, H. (2020). Job scheduling for large-scale machine learning clusters. In D. Han & A. Feldmann (Eds.), *CoNEXT '20: The 16th International Conference on emerging Networking EXperiments and Technologies, Barcelona, Spain, December, 2020* (pp. 108–120). ACM. https://doi.org/10.1145/3386367.3432588 (see p. 72).

Wang, S., & Kanwar, P. (2019). *BFloat16: The secret to high performance on Cloud TPUs*. Retrieved February 1, 2021, from https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus (see p. 92).

Wang, S., Zhang, M., Li, G., Liu, C., Liu, Y., Jia, X., & Xu, M. (2021). Making Multi-String Pattern Matching Scalable and Cost-Efficient with Programmable Switching ASICs. *40th IEEE Conference on Computer Communications, INFOCOM 2021, Vancouver, BC, Canada, May 10-13, 2021*, 1–10. https://doi.org/10.1109/INFOCOM42981.2021.9488796 (see p. 52).

Ware, R., Mukerjee, M. K., Seshan, S., & Sherry, J. (2019). Modeling BBR's Interactions with Loss-Based Congestion Control. *Proceedings of the Internet Measurement Conference, IMC 2019, Amsterdam, The Netherlands, October 21-23, 2019*, 137–143. https://doi.org/10.1145/3355369.3355604 (see pp. 2, 53, 192, 199).

Wetherall, D. (2002). Active Network Vision and Reality: Lessons from a Capsule-Based System. *2002 DARPA Active Networks Conference and Exposition (DANCE 2002), 29-31 May 2002, San Francisco, CA, USA*, 25. https://doi.org/10.1109/DANCE.2002.1003482 (see p. 14).

Wetherall, D., & Tennenhouse, D. L. (2019). Retrospective on "towards an active network architecture". *Comput. Commun. Rev.*, *49*(5), 86–89. https://doi.org/10.1145/3371934.3371961 (see p. 29).

Wetherall, D., Guttag, J., & Tennenhouse, D. (1998). ANTS: a toolkit for building and dynamically deploying network protocols. *1998 IEEE Open Architectures and Network Programming*, 117–129. https://doi.org/10.1109/OPNARC.1998.662048 (see p. 14).

Wiering, M. (1999, February 17). *Explorations in Efficient Reinforcement Learning* [PhD Thesis]. University of Amsterdam, Amsterdam. http://hdl.handle.net/11245/1.164863 (see p. 89).

Wiles, K. (2021). *Pktgen-DPDK – DPDK-based packet generator*. Retrieved May 29, 2021, from https://github.com/pktgen/Pktgen-DPDK (see pp. 44, 174).

Williams, R. J. (1992). Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Mach. Learn.*, *8*, 229–256. https://doi.org/10.1007/BF00992696 (see p. 88).

Wolf, T., & Turner, J. S. (2001). Design issues for high-performance active routers. *IEEE J. Sel. Areas Commun.*, *19*(3), 404–409. https://doi.org/10.1109/49.917702 (see pp. 15, 26, 31).

Wu, X., Guo, W., Wei, H., & Xing, X. (2021). Adversarial Policy Training against Deep Reinforcement Learning. In M. Bailey & R. Greenstadt (Eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (pp. 1883–1900). USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/wu-xian (see p. 100).

Xi, Z., Pang, R., Ji, S., & Wang, T. (2021). Graph Backdoor. In M. Bailey & R. Greenstadt (Eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (pp. 1523–1540). USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/xi (see p. 103).

Xiang, C., Bhagoji, A. N., Sehwag, V., & Mittal, P. (2021). PatchGuard: A Provably Robust Defense against Adversarial Patches via Small Receptive Fields and Masking. In M. Bailey & R. Greenstadt (Eds.), *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (pp. 2237–2254). USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/xiang (see p. 102).

Xie, G., Li, Q., Jiang, Y., Dai, T., Shen, G., Li, R., Sinnott, R. O., & Xia, S. (2020). SAM: Self-Attention based Deep Learning Method for Online Traffic Classification. In B. Arzani & X. Jin (Eds.), *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM, Virtual Event, USA, August 14, 2020* (pp. 14–20). ACM. https://doi.org/10.1145/3405671.3405811 (see p. 64).

Xie, S., Davidson, S., Magaki, I., Khazraee, M., Vega, L., Zhang, L., & Taylor, M. B. (2018). Extreme Datacenter Specialization for Planet-Scale Computing: ASIC Clouds. *ACM SIGOPS Oper. Syst. Rev.*, *52*(1), 96–108. https://doi.org/10.1145/3273982.3273991 (see p. 92).

Xilinx. (2021a). *Xilinx Revolutionizes the Modern Data Center with Software-Defined, Hardware Accelerated Alveo SmartNICs*. Retrieved May 11, 2021, from https://www.xilinx.com/news/press/2021/xilinx-revolutionizes-the-modern-data-center-with-software-defined-hardware-accelerated-alveo-smartnics.html (see p. 26).

Xilinx. (2021b, March 11). *AMD OpenNIC Project*. Retrieved March 10, 2022, from https://github.com/Xilinx/open-nic (see p. 26).

Xin, L. (2021, June 4). *An easier way to go: SCTP over UDP in the Linux kernel*. Retrieved February 25, 2022, from https://developers.redhat.com/articles/2021/06/04/easier-way-go-sctp-over-udp-linux-kernel (see p. 22).

Xiong, Z., & Zilberman, N. (2019). Do Switches Dream of Machine Learning?: Toward In-Network Classification. In *Proceedings of the 18th*

ACM Workshop on Hot Topics in Networks, HotNets 2019, Princeton, NJ, USA, November 13-15, 2019 (pp. 25–33). ACM. https://doi.org/10.1145/3365609.3365864 (see pp. 54, 187).

Xu, X., Wang, Q., Li, H., Borisov, N., Gunter, C. A., & Li, B. (2021). Detecting AI Trojans Using Meta Neural Analysis. *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, 103–120. https://doi.org/10.1109/SP40001.2021.00034 (see p. 104).

Xu, Z., Tang, J., Yin, C., Wang, Y., & Xue, G. (2019). Experience-Driven Congestion Control: When Multi-Path TCP Meets Deep Reinforcement Learning. *IEEE J. Sel. Areas Commun.*, *37*(6), 1325–1336. https://doi.org/10.1109/JSAC.2019.2904358 (see p. 65).

Yang, L., Anderson, T. A., Gopal, R., & Dantu, R. (2004). Forwarding and Control Element Separation (ForCES) Framework. https://doi.org/10.17487/RFC3746 (see p. 18).

Yap, K., Motiwala, M., Rahe, J., Padgett, S., Holliman, M. J., Baldus, G., Hines, M., Kim, T., Narayanan, A., Jain, A., Lin, V., Rice, C., Rogan, B., Singh, A., Tanaka, B., Verma, M., Sood, P., Tariq, M. M. B., Tierney, M., … Vahdat, A. (2017). Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering. *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM 2017, Los Angeles, CA, USA, August 21-25, 2017*, 432–445. https://doi.org/10.1145/3098822.3098854 (see p. 21).

Yau, D. K. Y., Lui, J. C. S., Liang, F., & Yam, Y. (2005). Defending against distributed denial-of-service attacks with max-min fair server-centric router throttles. *IEEE/ACM Trans. Netw.*, *13*(1), 29–42. https://doi.org/10.1109/TNET.2004.842221 (see p. 135).

Yin, X., Jindal, A., Sekar, V., & Sinopoli, B. (2015). A Control-Theoretic Approach for Dynamic Adaptive Video Streaming over HTTP. In S. Uhlig, O. Maennel, B. Karp & J. Padhye (Eds.), *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2015, London, United Kingdom, August 17-21, 2015* (pp. 325–338). ACM. https://doi.org/10.1145/2785956.2787486 (see p. 70).

Yun, X., Wang, Y., Zhang, Y., & Zhou, Y. (2016). A Semantics-Aware Approach to the Automated Network Protocol Identification. *IEEE/ACM Trans. Netw.*, *24*(1), 583–595. https://doi.org/10.1109/TNET.2014.2381230 (see p. 64).

Zerwas, J., Kalmbach, P., Henkel, L., Rétvári, G., Kellerer, W., Blenk, A., & Schmid, S. (2019). NetBOA: Self-Driving Network Benchmarking. *Proceedings of the 2019 Workshop on Network Meets AI & ML, NetAI@SIGCOMM 2019, Beijing, China, August 23, 2019*, 8–14. https://doi.org/10.1145/3341216.3342207 (see p. 64).

Zhang, H., Chen, H., Xiao, C., Gowal, S., Stanforth, R., Li, B., Boning, D. S., & Hsieh, C. (2020). Towards Stable and Efficient Training of

Verifiably Robust Neural Networks. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net. https://openreview.net/forum?id=Skxuk1rFwB (see p. 101).

Zhang, J., Guo, Z., Ye, M., & Chao, H. J. (2020). SmartEntry: Mitigating Routing Update Overhead with Reinforcement Learning for Traffic Engineering. In B. Arzani & X. Jin (Eds.), *Proceedings of the 2020 Workshop on Network Meets AI & ML, NetAI@SIGCOMM, Virtual Event, USA, August 14, 2020* (pp. 1–7). ACM. https://doi.org/10.1145/3405671.3405809 (see p. 62).

Zhang, K., Zhuo, D., & Krishnamurthy, A. (2020). Gallium: Automated Software Middlebox Offloading to Programmable Switches. In H. Schulzrinne & V. Misra (Eds.), *SIGCOMM '20: Proceedings of the 2020 Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication, Virtual Event, USA, August 10-14, 2020* (pp. 283–295). ACM. https://doi.org/10.1145/3387514.3405869 (see p. 46).

Zhang, M., Dusi, M., John, W., & Chen, C. (2009). Analysis of UDP Traffic Usage on Internet Backbone Links. *Ninth Annual International Symposium on Applications and the Internet, SAINT 2009, Bellevue, Washington, USA, July 20-24, 2009, Proceedings*, 280–281. https://doi.org/10.1109/SAINT.2009.65 (see p. 215).

Zhao, B. Z. H., Agrawal, A., Coburn, C., Asghar, H. J., Bhaskar, R., Kâafar, M. A., Webb, D., & Dickinson, P. (2021). On the (In)Feasibility of Attribute Inference Attacks on Machine Learning Models. *CoRR, abs/2103.07101*. https://arxiv.org/abs/2103.07101 (see p. 105).

Zhao, Y., Yang, K., Liu, Z., Yang, T., Chen, L., Liu, S., Zheng, N., Wang, R., Wu, H., Wang, Y., & Zhang, N. (2021). LightGuardian: A Full-Visibility, Lightweight, In-band Telemetry System Using Sketchlets. In J. Mickens & R. Teixeira (Eds.), *18th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2021, April 12-14, 2021* (pp. 991–1010). USENIX Association. https://www.usenix.org/conference/nsdi21/presentation/zhao (see p. 50).

Zhou, Q., Guo, S., Qu, Z., Guo, J., Xu, Z., Zhang, J., Guo, T., Luo, B., & Zhou, J. (2021). Octo: INT8 Training with Loss-aware Compensation and Backward Quantization for Tiny On-device Learning. In I. Calciu & G. Kuenning (Eds.), *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021* (pp. 177–191). USENIX Association. https://www.usenix.org/conference/atc21/presentation/zhou-qihua (see p. 93).

Zhu, H., Gupta, V., Ahuja, S. S., Tian, Y., Zhang, Y., & Jin, X. (2021). Network planning with deep reinforcement learning. In F. A. Kuipers & M. C. Caesar (Eds.), *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021* (pp. 258–271). ACM. https://doi.org/10.1145/3452296.3472902 (see p. 73).

Zilberman, N., Audzevich, Y., Covington, G. A., & Moore, A. W. (2014). NetFPGA SUME: Toward 100 Gbps as Research Commodity. *IEEE Micro*, *34*(5), 32–41. https://doi.org/10.1109/MM.2014.61 (see p. 26).