

Li, Xicheng (2026) *Accelerating network services with programmable data planes.* PhD thesis.

https://theses.gla.ac.uk/86369/

Copyright and moral rights for this work are retained by the author

A copy can be downloaded for personal non-commercial research or study, without prior permission or charge

This work cannot be reproduced or quoted extensively from without first obtaining permission from the author

The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the author

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given

Enlighten: Theses
https://theses.gla.ac.uk/
research-enlighten@glasgow.ac.uk

Accelerating Network Services with Programmable Data Planes

Xicheng Li

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 2026

© Xicheng Li

Abstract

Programmable data planes allow network functions to be offloaded onto the forwarding path, potentially avoiding the latency and coordination overhead of processing packets in control-plane or end-host software. The open challenge is to identify the resource and processing costs, and the restrictions on service behaviour, required for a network function to operate within the data plane without runtime control-plane or user-space intervention, and how to retain a fast packet-processing path when its logic requires cross-packet history or external context.

This thesis investigates this question through two contrasting arrangements of runtime state maintenance, and uses a framework based on state ownership and data-plane independence to compare their costs and constraints.

Programmable Dataplane (PDP) technologies like P4 and *Extended Berkeley Packet Filter* (eBPF) each have inherent limitations: P4's pipeline model, which varies by target, constrains arithmetic operations and state management, and may impose per-packet execution budgets; eBPF, while more flexible as it runs on the end host, is subject to verifier-imposed execution constraints.

To address these challenges, the thesis designs and evaluates a *Domain Name System* (DNS) cache implemented entirely in a P4 pipeline on the *Behavioral Model v2* (BMv2) software switch, and develops an eBPF/*eXpress Data Path* (XDP) architecture for IEC 61850 *Generic Object Oriented Substation Event* (GOOSE) anomaly detection in *Industrial Control Systems* (ICS) with user-space coordination and digital twin integration.

The DNS cache performs lookup and update without per-packet control-plane intervention, serving hits locally and forwarding misses or refresh queries upstream. On BMv2, it supports bounded-length A-record lookups, uses recirculation, and allocates approximately 78 KiB of register state for cache and epoch storage. An indicative comparison reports approximately 32 % fewer parser states visited per packet with binary-decomposition padding removal on recirculation, compared with a byte-loop implementation. Its admission and freshness trade-offs illustrate the service restrictions of maintaining state entirely within the data plane.

The ICS architecture validates GOOSE sequence progression in XDP, delegates anomaly aggregation and deny-list updates to user space, and provides a co-simulation interface to physical-process context. In the evaluated *Virtual Ethernet* (veth) testbed, the XDP detection path has a median per-packet latency of 3.815 μs, compared with 2.766 μs for a no-op XDP baseline. Mean reaction intervals from the threshold-triggering anomaly to the first subsequently blocked frame are 116 and 127 ms for two sequence-violation attack classes, including event delivery, the user-space map update, and the wait for the next frame. These results support the feasibility of the detection and enforcement paths.

Taken together, the prototypes show that software-maintained policy can support an independent packet-processing path, while maintaining state entirely within the data plane imposes resource costs and service restrictions. These trade-offs help guide the selection of an offloading mechanism.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my supervisor, Prof. Dimitrios Pezaros, for his continued support of my research, academic development, and career aspirations.

I thank my review panel, Emma Li, Lewis Mackenzie and Colin Perkins, for their advice and guidance during the annual progression reviews.

My thanks also go to my viva panel, Charalampos Rotsos, Burak Kizilkaya and Mathieu Chollet, for their careful reading of this thesis and constructive feedback during the viva.

I thank my colleagues and friends in NetLab and the Cyber Defence Lab for their research discussions, technical assistance, and the many enjoyable conversations over a pint.

I also thank the SoCS IT support team for their help with network connectivity in our G131 machine room.

Special thanks to Kai Feng, Zhanyan Qiu, and my neighbours in G161—Zeyan Liang and Zhuoran Tan—for their encouragement, emotional support, and all the laughter we shared, and to Yiwen Shang, Yongting Chen, and Puqi Zhou for our annual Zoomposiums. My thanks extend to all the friends I have not named here, whose kindness and company have meant so much to me throughout this journey.

I am also grateful for the Cosener's workshops, which offered valuable research insights and opportunities to connect with peers. I especially cherish the discussions we shared and the friendships I formed there with Weihe Li, Kaiyi Zhang, Damu Ding and Wenrui Zuo.

Finally, I thank my parents for their unwavering emotional and financial support, which enabled me to pursue this degree and research an area I am passionate about.

Publications

Work presented in this thesis has appeared in the following publications:

- Li, X., & Pezaros, D. (2025). Dataplane-Only DNS Caching in P4. *2025 IEEE Symposium on Computers and Communications (ISCC)*, 1–6. https://doi.org/10.1109/ISCC65549.2025.11326441
- Holik, F., Cook, M. M., Li, X., Shah, A. A., & Pezaros, D. (2025). Programmable Data Planes for Increased Digital Resilience in OT Networks. *IEEE Communications Magazine*, *63*(7), 162–169. https://doi.org/10.1109/MCOM.001.2400446
- Evripidou, S., Li, X., Alquran, M., Cheng, R., Taha, A., Imran, M., Flynn, D., & Pezaros, D. (2026). Organisational Cybersecurity Challenges in Digital Twin Development: A Critical Analysis and Research Directions. *Computers & Security*, 105001
- Li, W., Li, X., Pezaros, D., & Patras, P. (2026). Charon: Stratified Priority Sampling for Differentiated Per-Flow Measurement in High-Speed Networks. *Proceedings of the ACM SIGCOMM 2026 Conference: Posters and Demos*

Contents

Abstract *i*

Acknowledgements *ii*

Publications *iii*

List of Tables *viii*

List of Figures *ix*

Acronyms *xi*

1 Introduction *1*

1.1 Overview . . . *1*

1.2 Background . . . *1*

1.3 Research Questions . . . *3*

1.4 Thesis Statement . . . *4*

1.5 Contributions . . . *4*

1.6 Thesis Outline and Structure . . . *5*

2 Programmable Data Planes *7*

2.1 Overview . . . *7*

2.2 Key Concepts and Terminology . . . *7*

2.3 Transition to SDN and Beyond . . . *8*

2.4 Modern Programmable Data Planes . . . *9*

2.4.1 P4 . . . *10*

2.4.2 Fast Software Packet Processing . . . *14*

2.4.3 eBPF . . . *15*

2.4.4 Comparative Analysis . . . *17*

2.5 A State-Ownership Framework for In-Network Offloading . . . *21*

2.5.1 Data-Plane State Ownership . . . *21*

2.5.2 Shared State Ownership . . . *21*

2.6 Application Domains and Research Gaps . . . *22*

2.6.1 In-Network Caching and Protocol-Specific Services . . 22
2.6.2 Traffic Monitoring and In-Network Intelligence 24
2.7 Summary 29

3 DNS Caching with Data-Plane State Ownership ***31***
3.1 Overview 31
3.2 Background 32
3.3 Motivation 32
3.4 Design and Implementation 34
3.4.1 System Architecture 34
3.4.2 Cache Design 36
3.4.3 Implementation 39
3.5 Design Choices for Data-Plane DNS Caching 41
3.5.1 Storage Primitive 42
3.5.2 Parser Strategy 42
3.5.3 Collision Handling 44
3.5.4 Cache Update Model 45
3.6 Evaluation 46
3.6.1 Experimental Setup 47
3.6.2 Resource Usage 48
3.6.3 Cache Hit Ratio 49
3.6.4 Processing Cost Analysis 50
3.6.5 Latency Analysis 51
3.7 Discussion and Summary 52
3.7.1 Lessons for P4 Function Development 52
3.7.2 Limitations 53
3.7.3 From Data-Plane to Shared State Ownership 53
3.7.4 Summary 54

4 GOOSE Detection with Shared State Ownership ***55***
4.1 Overview 55
4.2 Motivation 56
4.3 GOOSE Communication and Detection Background 57
4.3.1 Protocol Role in the Explored Scenario 57
4.3.2 Protocol State and Prior Detection Approaches 58
4.4 Threat Model 58
4.5 Function Design 59
4.5.1 GOOSE Analysis and Policy Maintenance 59
4.5.2 Use Case: Integration with Transport–Energy Co-Simulation 61
4.6 Implementation 62

4.6.1 Prototype Environment 62
4.6.2 Experimental Data and Transmission Conditions . . . 63
4.6.3 In-Kernel Fast-Path Detection 64
4.6.4 User-Space Orchestration 65
4.7 Attack Walk-Throughs 67
4.7.1 Denial of Service 67
4.7.2 False Data Injection 67
4.7.3 Replay Attack 68
4.8 Architectural Comparison and Indicative Measurements 68
4.9 Discussion and Summary 71
4.9.1 State Ownership and Function Partitioning 72
4.9.2 Data-Plane Independence and Enforcement Delay . . 72
4.9.3 External Context and Policy Design 72
4.9.4 Scope and Remaining Limitations 72

5 Guidelines for Developing Stateless and Stateful Applications 75
5.1 Lessons Learnt from the Two Systems 75
5.1.1 Jointly Design State Representation and Packet Processing 75
5.1.2 Separate Immediate Decisions from Policy Maintenance 75
5.1.3 Distinguish Execution Bounds from Measured Performance 76
5.2 Guidelines for Application Development 76

6 Conclusion 77
6.1 Thesis Statement Revisited 77
6.2 Future Directions 78
6.2.1 Packet-Carried Context Within Programmable Domains 78
6.2.2 Selective In-Network Processing With a Placement Advantage 79
6.3 Concluding Remarks 79

References 81

List of Tables

2.1 Comparison of P4 and eBPF across five dimensions. 20

2.2 Implementations mapped to the state-ownership framework. 27

3.1 Static P4 resource usage for the register-based and MAT-based DNS cache implementations on BMv2. 48

3.2 Probe-depth distribution for insertions and warm lookups under the two-probe linear scheme, including the probabilistic bypass fraction. 49

3.3 Pipeline traversals and parser-state visits per packet. 50

3.4 Local cache-hit latency on BMv2 (ms). 51

4.1 Sequence-number violation conditions for GOOSE frames. . 64

4.2 BPF map allocations of the proposed in-kernel GOOSE detection. 69

4.3 Per-packet latency (ns) of `goose_xdp` vs. a no-op XDP baseline. 71

4.4 Indicative mitigation timings for attacks crossing the replay-class threshold ($R = 3$). 71

List of Figures

1.1 Thesis roadmap: research questions, statement claims, contributions, and chapters. 5

2.1 P4 compilation and runtime workflow, reproduced from P4.org. 10

2.2 P4 pipeline architecture of V1model. 12

2.3 XDP packet-processing pipeline. 16

3.1 DNS caching within the BMv2 v1model architecture. 35

3.2 Data flow through the P4 ingress pipeline. 35

3.3 Bit layout of a 64-bit DNS cache slot. 36

3.4 Wire-format layout of a DNS query for `google.com`. 39

3.5 The single-host topology used for evaluation. 47

3.6 Cache occupancy of the register-based and MAT-based designs. 49

3.7 Local cache-hit latency CDF for the two cache designs. . . . 51

4.1 End-to-end GOOSE latency across handling paths on an emulated testbed (Holik, Cook *et al.*, 2025). 56

4.2 GOOSE data-plane analysis within the transport–energy co-simulation use case. 61

4.3 In-kernel GOOSE anomaly-detection pipeline. 64

4.4 GOOSE detector orchestration and co-simulation interface. . 66

4.5 XDP and SGFabric GOOSE detection pipelines. 69

List of Algorithms

1 Cache Insert / Update with Bounded Probing and Stale-Cell Reclamation 37

2 Cache Lookup with Bounded Probing, Bypass and Epoch Aging 38

3 DNS Cache Slot Eligibility for Insertion or Update 38

4 Data-plane DNS Parser & Answer Re-assembly 40

Acronyms

ACL Access Control List 1, 65

AI Artificial Intelligence 79

API Application Programming Interface 10, 12, 13, 18, 21, 32

ASIC Application-Specific Integrated Circuit 1, 2, 8, 13, 18–20, 29, 34

BER Basic Encoding Rules 60, 65, 68, 70

BMv2 Behavioral Model v2 i, 10, 11, 13, 18, 25, 31, 34, 43, 46–48, 53, 76

BPF Berkeley Packet Filter 15–17, 19, 21, 26, 59, 60, 63, 64, 66, 72, 73

CDN Content Delivery Network 32

CPU Central Processing Unit 11, 15, 16, 18, 20, 21, 25, 47, 63, 65, 70

CRC Cyclic Redundancy Check 44

CRC32 32-bit Cyclic Redundancy Check 34, 36, 44, 49, 50, 53

DAG Directed Acyclic Graph 17, 19

DERMS Distributed Energy Resource Management System 62

DMA Direct Memory Access 33

DNS Domain Name System i, 3, 4, 6, 21, 22, 24, 29–35, 37, 39, 41–43, 47, 48, 52–55, 72, 75–77

DNSSEC DNS Security Extensions 53

DoH DNS over HTTPS 32

DoS Denial of Service 25, 67

DoT DNS over TLS 32

DPDK Data Plane Development Kit 2, 10, 15, 26

DPU Data Processing Unit 11, 25

DRAM Dynamic Random-Access Memory 23, 54

DSCP Differentiated Services Code Point 25

DSL Domain-Specific Language 2, 9

DT Digital Twin 58

eBPF Extended Berkeley Packet Filter i, 2, 3, 6–8, 10, 15–22, 25, 26, 29, 30, 53, 55, 56, 58–62, 64, 66–69, 76, 78

EV Electric Vehicle 58, 59, 61, 62, 73

FPGA Field-Programmable Gate Array 8, 18, 26

FSM Finite State Machine 12, 19

GOOSE Generic Object Oriented Substation Event i, 4–6, 22, 26, 29, 30, 54–73, 75–78

ICS Industrial Control System i

IED Intelligent Electronic Device 57, 59, 63, 67

IETF Internet Engineering Task Force 2

INT In-band Network Telemetry 24, 25

IOAM In Situ Operations, Administration, and Maintenance 78

IP Internet Protocol 24, 32, 33, 35, 37, 45, 52

IPU Infrastructure Processing Unit 2

IT Information Technology 22

JIT Just-in-Time 15, 17

JSON JavaScript Object Notation 66

LAN Local Area Network 56, 58, 67

LFU Least Frequently Used 45, 46

LPM Longest-Prefix Match 12

LRU Least Recently Used 16, 24, 45, 46

MAC Medium Access Control 1, 60, 63, 64, 67, 68, 70, 73

MAT Match-Action Table 47–51

MAU Match-Action Unit 42

MTU Maximum Transmission Unit 78

NAT Network Address Translation 23

NFP Netronome Flow Processor 18

NIC Network Interface Card 9, 15, 18, 60, 64

NOS Network Operating System 2

OT Operational Technology 22, 26, 30

PDP Programmable Dataplane i, 7

PDU Protocol Data Unit 60

PHV Packet Header Vector 12, 14, 43

PISA Protocol Independent Switch Architecture 13

PNA Portable NIC Architecture 18

PSA Portable Switch Architecture 11, 13

QoS Quality of Service 2

RAM Random Access Memory 47, 69

RAN Radio Access Network 9

RDMA Remote Direct Memory Access 23, 24, 54

RFC Request for Comments 2

RMT Reconfigurable Match Tables 2, 9

RPC Remote Procedure Call 25

SDE Software Development Environment 10, 11, 13

SDN Software-Defined Networking 2, 5, 7–9, 34, 56, 58

SRAM Static Random-Access Memory 12, 16, 18, 29, 42, 45, 52, 53

SUMO Simulation of Urban MObility 61, 63, 66

TC Traffic Control 16, 18, 25

TCAM Ternary Content-Addressable Memory 12, 16, 42

TCP Transmission Control Protocol 26, 32, 57, 61, 62

TLS Transport Layer Security 32

TLV Type-Length-Value 53, 60, 68

TNA Tofino Native Architecture 2, 11, 13, 43

TraCI Traffic Control Interface 61, 63, 66

TTL Time-to-Live 32, 33, 37, 41, 45, 46, 50, 52, 53, 77

UDP User Datagram Protocol 31, 32, 34, 39, 41

UI User Interface 62

UTC Coordinated Universal Time 57

veth Virtual Ethernet i, 63, 70

VLAN Virtual Local Area Network 60, 63

VNF Virtual Network Function 58, 66

XDP eXpress Data Path i, 6, 8, 15, 16, 18, 19, 25, 55, 57, 59, 60, 62–64, 66–70, 72, 78

Chapter 1

Introduction

1.1 Overview

Programmable data planes allow network services to execute directly on the packet-forwarding path, potentially reducing processing latency and software coordination overhead (Michel *et al.*, 2021). However, their execution and resource constraints limit the service behaviour they can support independently. The availability of programmable switches and in-kernel packet-processing frameworks creates opportunities to investigate how these limitations can be addressed (Bosshart *et al.*, 2014; Vieira *et al.*, 2020).

This thesis addresses how to organise network-service processing and state maintenance under these constraints while retaining an independent packet-processing fast path. Using two contrasting systems, it relates service requirements to the capabilities of the target platform and examines the trade-offs between self-contained data-plane processing and coordination with end-host software. The resulting evidence informs offloading decisions in terms of state ownership and data-plane independence.

The remainder of this chapter presents the background (Section 1.2), research questions and thesis statement (Sections 1.3 and 1.4), contributions (Section 1.5), and thesis structure (Section 1.6).

1.2 Background

Historically, the data plane (and particularly the network switch) was implemented as an *Application-Specific Integrated Circuit* (ASIC) whose forwarding functionality was fixed by the vendor at design time, built to forward packets at line rate. A layer-2 switch, for example, learns *Medium Access Control* (MAC)-to-port mappings from incoming source addresses and forwards frames according to their destination, while later multi-layer switches added layer-3 processing together with features such as *Access Control List*

(ACL) and *Quality of Service* (QoS). Differences between device generations often concerned resources such as table sizes and buffer capacity. Nevertheless, the forwarding pipeline remained hardwired in silicon; only the table entries and configuration parameters could be modified at runtime (Bosshart *et al.*, 2013). In this sense, the switch could only execute the set of forwarding behaviours that its vendor had built in, preventing operators from freely defining arbitrary header parsing or complex stateful packet modifications (Sivaraman *et al.*, 2020).

The concept of *Software-Defined Networking* (SDN) was subsequently proposed to decouple the control plane from the data plane (also referred to as the forwarding plane) (Feamster *et al.*, 2014; Kreutz *et al.*, 2014), and OpenFlow emerged as the first widely adopted realisation of this idea (McKeown *et al.*, 2008). OpenFlow exposed existing match-action capabilities through an open, standardised southbound interface, instructing the data plane to match incoming packets against a set of flow rules and to perform supported actions such as forwarding, dropping, or header modification. Crucially, this interface abstracts away the heterogeneity of the underlying hardware across vendors and platforms, allowing the control plane to maintain a centralised view of the network and to update forwarding rules at runtime (Michel *et al.*, 2021). This abstraction has catalysed a wave of innovations in network management, traffic engineering (Ferguson *et al.*, 2021), and security (Scott-Hayward *et al.*, 2015), and has paved the way for the next generation of programmable data planes (Amin *et al.*, 2018).

However, early OpenFlow remained tied to protocol-defined header fields and actions supported by the underlying hardware, leaving the forwarding pipeline largely non-programmable (Bosshart *et al.*, 2013; 2014). Large-scale cloud operators became increasingly unwilling to tolerate vendor lock-in, the high cost of purpose-built hardware, and the inflexibility of fixed pipelines that prevented them from deploying custom network functions and new protocols (Ferguson *et al.*, 2021; Jouet & Pezaros, 2017; Singh *et al.*, 2015). As a result, demand for fully programmable (or at least runtime-reconfigurable) data planes grew rapidly (Jouet & Pezaros, 2017).[1] This demand gave rise to the notion of white-box switching and modern *Network Operating System* (NOS) platforms in the early 2010s, followed from the mid-2010s onwards by the development of high-level *Domain-Specific Languages* (DSLs) and renewed traction of programmable host-stack technologies (Zilberman *et al.*, 2015); notable examples include P4 (Bosshart *et al.*, 2014), *Extended Berkeley Packet Filter* (eBPF) (Vieira *et al.*, 2020), *Data Plane Development Kit* (DPDK) (Abranches *et al.*, 2022), and NetFPGA (Lockwood *et al.*, 2007; Zilberman *et al.*, 2014). On the hardware side, Intel's Tofino line of switching ASICs commercialised the *Reconfigurable Match Tables* (RMT) model (Bosshart *et al.*, 2013) with a *Tofino Native Architecture* (TNA) design (Agrawal & Kim, 2020).[2]

Among these developments, P4 and eBPF stand out as two complementary

[1] A similar situation can be observed in standardisation bodies such as the *Internet Engineering Task Force* (IETF), where a new protocol typically passes through several phases over many years to mature from an initial draft into a published *Request for Comments* (RFC) (Wirtgen *et al.*, 2023).

[2] Intel discontinued the Tofino product line in early 2023 and shifted its focus to the *Infrastructure Processing Unit* (IPU).

paradigms: P4 targets on-path switch hardware, whereas eBPF targets host-side software (Holik, Cook *et al.*, 2025; Kianpisheh & Taleb, 2022). This thesis accordingly focuses on them as the primary technologies under investigation.

1.3 Research Questions

The emergence of programmable data planes has expanded the scope of both network engineering and research. However, the heterogeneity of data-plane techniques has raised new challenges in the design of network functions (Michel *et al.*, 2021). The following three research questions develop one argument in successive stages. RQ1 establishes the costs and service restrictions of self-contained data-plane processing. RQ2 then asks how to organise processing when service requirements exceed the constraints of that model. RQ3 draws on both investigations to examine the trade-offs that guide offloading decisions.

RQ1. *What resource and processing costs, and what restrictions on service behaviour, arise when a network service that conventionally requires control-plane intervention is realised as a fully self-contained data-plane pipeline?*

Existing programmable data plane architectures such as P4 impose strict constraints on their pipelines, resulting in limited capabilities in stateful processing, arithmetic operations, and memory capacity (Hauser *et al.*, 2023). Moreover, control plane communication introduces additional latency and complexity into the design of network functions, and may complicate fault tolerance in tightly coupled systems (Kreutz *et al.*, 2014). These constraints make runtime state maintenance a central design challenge for independent data-plane processing. A *Domain Name System* (DNS) cache is a demanding case: its entry-expiry and refresh obligations normally fall to the control plane. Chapter 3 examines the costs of making such a service fully self-contained within a P4 pipeline, characterising pipeline complexity, storage requirements, per-packet processing, cache-hit latency, and the trade-offs in cache admission, invalidation, and refresh.

RQ2. *When service requirements exceed the constraints of a self-contained data-plane pipeline, how can processing and state maintenance be partitioned between the data plane and end-host software while retaining an independent packet-processing fast path?*

Commodity end-hosts provide general-purpose computation and memory for functions whose state maintenance or analysis exceeds the chosen data-plane implementation's capabilities (Ports & Nelson, 2019; Vieira *et al.*, 2020).

The coordination challenge is to make software-maintained state available to the fast path without making each packet wait for a software decision (Høiland-Jørgensen *et al.*, 2018; Van Tu *et al.*, 2019). Chapter 4 examines this through *Generic Object Oriented Substation Event* (GOOSE) anomaly detection, combining in-kernel checks with asynchronous user-space policy updates.

RQ3. *How do state ownership and data-plane independence, together with their associated costs and coordination requirements, inform network-function offloading decisions?*

RQ3 synthesises the costs and service restrictions investigated in RQ1 with the partitioning and coordination requirements investigated in RQ2. Chapter 5 draws on the development of the DNS cache and GOOSE detector to formulate guidelines for stateless and stateful applications. The guidelines connect service requirements, state-maintenance responsibilities, and processing dependencies to offloading decisions.

1.4 Thesis Statement

In response to these questions, this thesis asserts that:[3]

> For bounded services, programmable data planes can maintain runtime state and complete forwarding decisions independently, subject to resource and service constraints. (*s0*) User-space coordination can extend data-plane processing through shared state maintenance while retaining an independent fast path. (*s1*) Because these arrangements impose different costs and coordination requirements, this thesis proposes a framework based on state ownership and data-plane independence to inform offloading decisions. (*s2*)

[3] The three claims are labelled *s0*, *s1*, and *s2*, where s abbreviates *statement* and indices begin at zero. These markers are reused throughout the thesis when individual claims are revisited.

1.5 Contributions

Grouped according to the claims (*s0–2*) in the thesis statement, this thesis makes the following contributions:

C1: Costs and limits of data-plane state ownership (*s0*). An analysis of how a network service can maintain its runtime state entirely within the data plane, and of the resource costs and service restrictions this entails. The DNS cache provides design and empirical evidence linking state representation and maintenance choices to processing overhead and service behaviour (Chapter 3).

C2: Shared state ownership with an independent fast path (*S1*). A design approach that divides runtime state-maintenance responsibilities between the data plane and end-host software while retaining independent per-packet processing. The GOOSE detector demonstrates the feasibility of this division and characterises the coordination costs of asynchronous policy updates with an in-kernel data path (Chapter 4).

C3: State ownership and offloading trade-offs (*S2*). A synthesis of the two systems that distinguishes state-maintenance responsibilities from per-packet processing dependencies. The resulting guidelines for stateless and stateful applications relate resource costs, service restrictions, and coordination requirements to the choice of an offloading arrangement (Chapter 5).

1.6 Thesis Outline and Structure

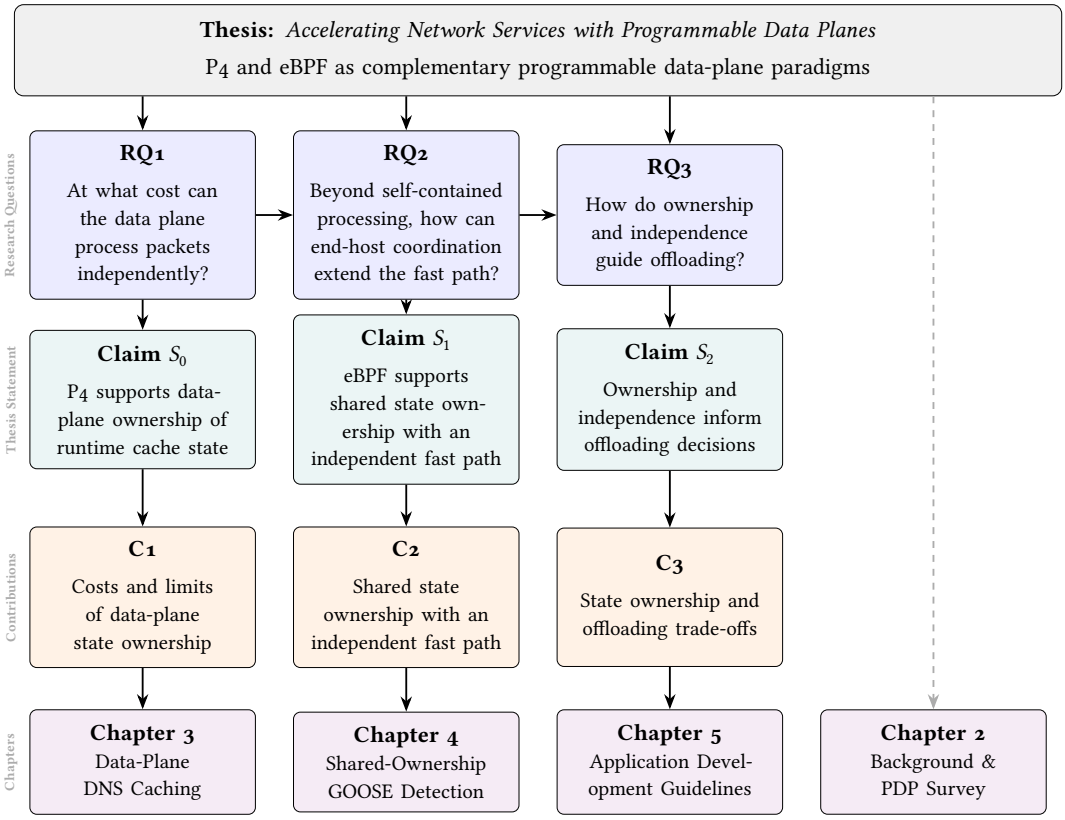


Figure 1.1: Thesis roadmap: research questions, statement claims, contributions, and chapters.

As illustrated in Figure 1.1, each research question is addressed by a dedicated chapter. The chapters are organised as follows:

Chapter 2 provides the technical background and literature review on programmable data planes, including the evolution from SDN/OpenFlow

to modern paradigms such as P4 and eBPF, and introduces a state-ownership framework for in-network offloading.

Chapter 3 presents the design and evaluation of a P4-based DNS cache that maintains its runtime state within the pipeline, characterising the resource and processing costs, and the service trade-offs, of control-plane-free offloading for simple key-value services.

Chapter 4 presents an eBPF/*eXpress Data Path* (XDP)-based GOOSE anomaly detection architecture and prototype with user-space policy maintenance, and outlines the engineering requirements for production deployment.

Chapter 5 presents development guidelines for stateless and stateful applications informed by lessons from both systems. It answers RQ3 by relating state ownership and processing dependencies to their costs and service restrictions.

Chapter 6 revisits the thesis statement and the scope of the findings, outlines future research directions, and offers concluding remarks.

Chapter 2

Programmable Data Planes

2.1 Overview

From circuit switching to packet switching, and from fat-tree to spine-leaf architectures, networking technologies have continuously evolved over the past several decades, underpinning the robust and scalable Internet infrastructure on which contemporary services depend (Feamster *et al.*, 2014).

This chapter provides a general overview of the principles underpinning *Software-Defined Networking* (SDN) and *Programmable Dataplanes* (PDPs), exploring their origins, conceptual frameworks, and interplay. Section 2.2 defines the key concepts and terminology used throughout the thesis. Section 2.3 begins with an overview of OpenFlow, an early and widely adopted standardisation effort that decouples the control plane from the data plane and introduces the match-action abstraction for forwarding, with a multi-table pipeline added in subsequent revisions. Section 2.4 examines P4, introduces software fast packet-processing approaches, and then discusses *Extended Berkeley Packet Filter* (eBPF), before comparing their execution models and capabilities. Section 2.5 sets out the state-ownership framework used to analyse prior work and the systems developed in subsequent chapters. Section 2.6 surveys related work organised by application domain, identifying the research gaps addressed by this thesis.

2.2 Key Concepts and Terminology

Network function is a logical function that implements a network service or set of packet-processing operations, such as forwarding, load balancing, or firewalling. It may be realised in hardware, software, or a combination of both (Halpern & Pignataro, 2015).

Data plane is the part of a network system responsible for packet processing and forwarding, including parsing, header modification, and

per-packet forwarding decisions (Haleplidis *et al.*, 2015). It can be implemented in hardware (e.g. *Application-Specific Integrated Circuits* (ASICs) or *Field-Programmable Gate Arrays* (FPGAs)) or software (e.g. the Linux networking stack or user-space packet-processing frameworks).

Control plane establishes and updates the forwarding rules and policies used by the data plane. It may be distributed across network devices or logically centralised, as in SDN (Haleplidis *et al.*, 2015). Switch-local control-plane software configures the device's pipeline through runtime interfaces and may communicate with an external controller.

User space is the execution environment for application processes outside the operating-system kernel. User-space programs access kernel services through interfaces such as system calls.

Kernel space is the privileged execution environment of the operating-system kernel, including its networking stack and in-kernel packet-processing mechanisms such as eBPF/*eXpress Data Path* (XDP).

State refers here to runtime information retained between packet-processing events and used to determine subsequent behaviour, including cache entries, sequence history, epoch counters, and enforcement policy. Per-packet metadata is ephemeral information whose lifetime is bounded by an individual packet's processing lifecycle.

Stateless describes processing that does not depend on runtime history retained from earlier packets or requests. Fixed configuration may still guide packet handling. For example, a set of hard-coded forwarding rules is stateless.

Stateful describes processing that depends on runtime state retained across packets or requests, such as cache contents or connection history.

State ownership identifies the components responsible for creating, updating, invalidating, and coordinating runtime state: the data plane, control-plane software, user-space software, or a combination of these. Ownership denotes maintenance responsibility, not physical storage location or read access.

Data-plane independence is the ability of the data plane to complete per-packet forwarding decisions without synchronous runtime intervention from software outside the data-plane implementation, such as a SDN controller.

2.3 Transition to SDN and Beyond

For decades, packet-switched networks were dominated by devices whose control logic and forwarding plane were tightly coupled within each vendor's

proprietary stack. Seeking greater flexibility and programmability, network engineers envisioned a centralised control plane that would act as the brain of the fabric, making forwarding decisions, performing real-time telemetry, and enforcing policies network-wide. This vision ultimately materialised as SDN.

Network operators had to configure devices individually through heterogeneous command-line interfaces, which led to inconsistent policies and considerable operational complexity. Management effort grew rapidly with the size of the fabric, and the absence of a global view impeded forwarding decisions that depend on network-wide state (Kreutz *et al.*, 2014).

Although some vendors sought to mitigate these frictions through proprietary management platforms, such solutions reinforced vendor lock-in and raised additional barriers to interoperability.

A further promise of SDN was therefore to open the control interface, fostering a broader ecosystem in which smaller vendors and the research community could contribute on a more equal footing.[1] A concrete step in this direction came from McKeown *et al.* (2008), whose OpenFlow protocol standardised the match-action forwarding model and exposed it to the controller through the flow table abstraction.

[1] A similar philosophy would later emerge in the telco domain as OpenRAN, which aims to disaggregate the *Radio Access Network* (RAN) stack and open up interfaces to enable multi-vendor deployments.

OpenFlow catalysed the adoption of software-defined networking, but its design exposed two related limitations as deployments matured. Its parser recognised only a fixed set of protocol headers defined by the specification, and its match-action pipeline was rigidly structured, offering little room for experimentation with new protocols or forwarding semantics (Bosshart *et al.*, 2014; Kreutz *et al.*, 2014).

In the following years, innovation came from both the hardware and software sides: reconfigurable match-action pipelines such as the *Reconfigurable Match Tables* (RMT) architecture (Bosshart *et al.*, 2013) relaxed the hardware rigidity, while the demand for protocol-independent programmability was answered by *P4: Programming Protocol-Independent Packet Processors* (Bosshart *et al.*, 2014), which operates at bit-string level and allows users to define their custom protocols and processing logic explicitly. These two strands of evolution, reconfigurable hardware pipelines and protocol-independent languages, underpin the programmable data planes surveyed in section 2.4.

2.4 Modern Programmable Data Planes

Building on the evolution traced in section 2.3, this section examines programmable switch pipelines alongside software approaches to efficient packet processing on commodity hosts.

P4 is a hardware-oriented *Domain-Specific Language* (DSL) tailored to line-rate match-action pipelines on programmable switches and *Network Inter-*

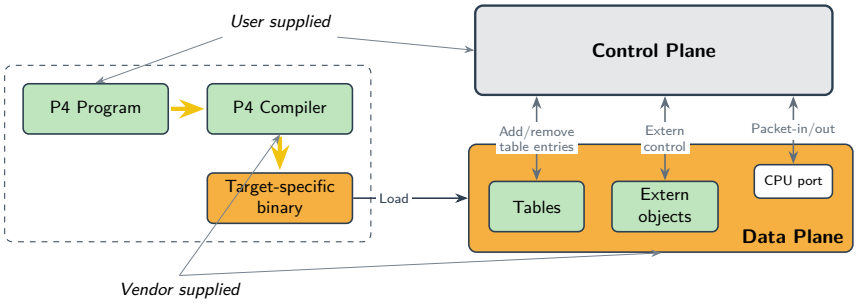


Figure 2.1: P4 compilation and runtime workflow, reproduced from P4.org.

face Cards (NICs), whereas eBPF is an in-kernel bytecode virtual machine that retrofits programmability into existing Linux networking subsystems on commodity devices (Hauser *et al.*, 2023; Michel *et al.*, 2021). Their different execution contexts provide complementary settings for examining packet-processing capabilities and state-maintenance responsibilities. Software acceleration also has a broader history, encompassing modular router architectures (Kohler *et al.*, 2000) and efficient packet I/O mechanisms provided by frameworks such as *Data Plane Development Kit* (DPDK) and netmap (DPDK Project, n.d.; Rizzo, 2012). Comparing P4 and eBPF clarifies the underlying architectural choices and their trade-offs, providing the technical background for the analytical framework in section 2.5.

Accordingly, section 2.4.1 examines P4, while section 2.4.2 introduces the software background for the eBPF discussion in section 2.4.3. Section 2.4.4 then compares P4 and eBPF.

2.4.1 P4

P4 targets hardware devices at line rate; this subsection examines its pipeline model, target architectures, and inherent constraints.

As illustrated in fig. 2.1, the user defines packet processing logic in a P4 program, which is then compiled by a target-specific compiler toolchain (also known as *Software Development Environment* (SDE)), provided either by the silicon vendor (e.g. the Intel P4 Studio SDE, which bundles the `bf-p4c` compiler for Tofino targets) or the open-source community (e.g. the `p4c` toolchain for *Behavioral Model v2* (BMv2) and other targets), into a target-specific binary, which is loaded onto the target device. The user can also interact with the target device at runtime through a control-plane *Application Programming Interface* (API), to populate match-action tables, read stateful objects such as counters and registers, and perform other target-specific operations.

The initial P4 specification, known as $P4_{14}$, is tightly coupled to the V1model architecture, which it implicitly assumes throughout the language (P4 Lan-

guage Consortium, 2016). Its successor, $P4_{16}$, decouples the language from any particular architecture by exposing the architecture and `extern` constructs, which delegate target-specific operational behaviours such as counters and registers to the architecture definition (P4 Language Consortium, 2024). This separation preserves the simplicity of the language primitives while enhancing portability and modularity.

The following terms are used frequently in the P4 ecosystem and are important to understand before delving into the details of P4 programming:

P4 language specification defines the syntax and semantics of the P4 programming language, including how to express packet parsing, match-action logic, and control flow. It is independent of any specific architecture or target implementation.

P4 architecture defines the programmable structure of the high-level packet processing pipeline, including the components (e.g. parsers, match-action units, deparsers) and their interconnections. It serves as a blueprint for how packets are processed within the data plane; three representative instances (V1model, *Portable Switch Architecture* (PSA), and *Tofino Native Architecture* (TNA)) are discussed below.

P4 target refers to the actual entity that executes the compiled P4 binary, which can be a software switch (e.g. BMv2), a programmable SmartNIC (e.g. AMD Pensando Elba *Data Processing Unit* (DPU)) or a programmable switch (e.g. Stordis or Edgecore switches based on Intel Tofino platform[2]). Each target may implement a specific P4 architecture, and is shipped with a vendor-specific SDE that compiles P4 programs into target-specific binaries.

[2] Intel discontinued the Tofino product line in 2023, but Tofino-based switches remain the de facto research platform for P4 hardware.

With these foundational concepts in place, the discussion now turns to the structural components that constitute a P4 pipeline.

Pipeline Model P4 programs describe packet processing as a sequential pipeline, composed of the following main components. Note that the exact composition and naming of these components are target-specific: for example, the V1model used by BMv2, as shown in fig. 2.2, does not include an ingress deparser or an egress parser, and `extern` components in hardware targets may exhibit different behaviours even if they appear identical or similar.

Ports, which are mapped to either virtual or physical network interfaces that accept Ethernet traffic. Special ports serve advanced functions and may be target-specific. Examples include the Central Processing Unit *(CPU) Port*, which sends packets to the control plane, and *Drop Port*, which discards traffic, similar to a null route (also known as a black hole) in the context of routing and switching.

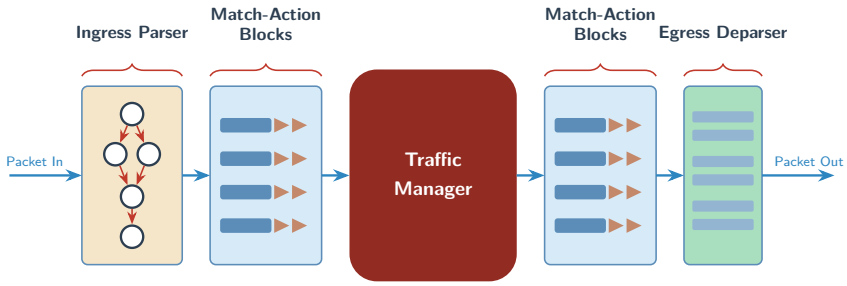


Figure 2.2: P4 pipeline architecture of V1model.

Parsers, which extract protocol headers from packets, are represented as *Finite State Machines* (FSMs) with two distinguished terminal states: `accept` and `reject`. They read packet headers field by field, enabling early rejection of invalid packets before further processing.

Match-Action Tables, which are the core building blocks of the packet processing pipeline, are normally configured from the control plane through target-specific APIs. In these tables, the target performs lookups on match fields using *Ternary Content-Addressable Memory* (TCAM) for ternary matches, *Static Random-Access Memory* (SRAM) (often together with hash-based exact-match structures) for exact matches, and target-specific lookup structures for *Longest-Prefix Match* (LPM) (Bosshart *et al.*, 2013; Hauser *et al.*, 2023), and applies associated actions to modify the packet, update its metadata, or set its egress port. The data used for comparison with the table entries may be extracted or derived from the packet.

Metadata, which holds ephemeral data generated or supplied during P4 program execution, is stored alongside parsed header fields in the *Packet Header Vector* (PHV) on PHV-based targets. It may carry intermediate values and processing decisions between pipeline components (i.e. control blocks), with its lifetime bounded by the packet's processing lifecycle.

Deparsers, which are placed at the end of the pipeline, reconstruct packets by prepending the processed headers to the payload and serialising the result into a raw byte stream before emitting it from the egress port or forwarding it to the traffic manager.

Target Architectures Different P4 architectures prescribe the composition and interconnection of the programmable pipeline; the choice of architecture determines both the available hardware features and the applicable programming constraints (Hauser *et al.*, 2023).

V1model.

V1model is the reference architecture for the BMv2 software switch and originates from the $P4_{14}$era, where it served as a simplified model used by BMv2 to approximate PISA. It defines a minimal pipeline comprising an ingress parser, an ingress match-action pipeline, a traffic manager, an egress match-action pipeline, and an egress deparser, as illustrated in fig. 2.2; notably, it omits an ingress deparser and an egress parser, and delegates checksum verification and computation to `extern` calls in the ingress and egress pipelines, respectively.

V1model is primarily intended for development and prototyping: it does not guarantee behavioural equivalence between software simulation and hardware targets, and therefore provides limited insight into the constraints encountered in real deployments. Hardware-oriented architectures, such as PSA and TNA, are consequently more representative of the production environments considered in this thesis.

PSA. PSA is a formal $P4_{16}$architecture that standardises the match-action pipeline model shared by reconfigurable hardware targets. In the words of its specification, *"the PSA is to the $P4_{16}$ language as the C standard library is to the C programming language"* (P4 Language Consortium, 2022): it defines a standard set of `extern` objects and pipeline interfaces without prescribing a specific hardware implementation. Compared with V1model, PSA introduces an ingress deparser and an egress parser at the traffic manager boundary, so that headers are explicitly serialised at the end of ingress processing and re-parsed at the start of egress processing. As a consequence, metadata does not carry over implicitly between the two pipelines; the programmer must explicitly re-populate any egress metadata through the egress parser, which enforces a clean separation of ingress and egress state.

TNA.

TNA is a proprietary $P4_{16}$architecture developed by Barefoot Networks (later Intel) for the Tofino family of programmable ASICs, which are widely adopted in both industry and academia (Hauser *et al.*, 2023; Intel Corporation, 2021). TNA extends the reconfigurable hardware pipeline model with a multi-pipeline design, allowing different P4 programs to be loaded onto distinct pipelines of the same switch simultaneously. It also provides a rich set of target-specific `externs` that expose hardware capabilities beyond the scope of the P4 language itself, including random value generation, packet timestamping, and in-data-plane packet generation. Programs targeting TNA must be compiled using the Intel P4 Studio SDE, which bundles the `bf-p4c` compiler and the BF Runtime API for control plane interaction.

Across these three architectures, the choice of target platform determines the set of available programming primitives and the ceiling of achievable programmability, which in turn shapes the set of in-network offloading patterns explored in this thesis.

Constraints and Limitations The capabilities of these P4 architectures are bounded by the target hardware, especially in their support for stateful processing, pipeline resources, and arithmetic.

Stateful processing.

Off-the-shelf P4 switches expose only tens of megabytes of on-chip memory shared across pipelines and stages (Hauser *et al.*, 2023; Zhu *et al.*, 2022), and the P4 specification disallows dynamic memory allocation, forcing programmers to size all stateful structures at compile time (Qian *et al.*, 2025). Recent work has sought to overcome these constraints by extending the pipeline with richer stateful primitives (Feng *et al.*, 2024; Scazzariello *et al.*, 2023). State-of-the-art approaches therefore employ probabilistic data structures, such as sketches (W. Li, Bütün *et al.*, 2025; W. Li, Chu *et al.*, 2025; Namkung *et al.*, 2022), to provide estimated statistics for high-volume traffic within limited memory budgets, while precise accounting remains infeasible.

Resource and pipeline constraints.

Certain P4 targets impose constraints on the number of PHV, the number of pipeline stages (Chen *et al.*, 2024), and execution time per packet (Bosshart *et al.*, 2013; Shirmarz *et al.*, 2025). These restrictions make it challenging to port network functions that involve complex operations. A common workaround is to recirculate packets or chain action blocks (Akem *et al.*, 2023; Kaplan & Feibish, 2022; Yoo & Chen, 2021); however, this introduces additional implementation complexity and reduces effective throughput.

Arithmetic restrictions.

P4 does not support floating-point arithmetic, although recent efforts have explored workarounds on specific hardware targets (Ding *et al.*, 2020; Yuan *et al.*, 2022). Fixed-point arithmetic can be employed within match-action tables to approximate floating-point operations, but this approach compromises both range and precision. Furthermore, certain primitives, such as bit slicing operators (`[ ]`), index expressions, data extraction, and object initialisers, require compile-time known values to access data within a packet.

2.4.2 Fast Software Packet Processing

Fast software packet processing on commodity hosts aims to achieve high throughput and low latency without using purpose-built hardware. This is typically achieved by optimising the packet I/O path and reducing the overhead of packet handling in kernel and user space, such as bypassing the kernel networking stack, avoiding per-packet socket buffer allocation, and minimising memory copies (DPDK Project, n.d.; Rizzo, 2012).

An earlier foundation for programmable software packet processing is Click, a modular software router architecture (Kohler *et al.*, 2000). It allows users

to build bespoke packet-processing pipelines by connecting individual elements, each implementing a specific function such as packet classification, queueing, scheduling, or forwarding. Click has also been used in commercial deployments, including early Cisco Meraki products.

DPDK provides libraries and drivers for efficient user-space packet processing. Its hardware poll-mode drivers typically access NIC receive and transmit queues directly, process packets in bursts, and bypass the conventional kernel networking stack. Direct hardware access requires a NIC supported by the selected poll-mode driver, and packet-processing threads are typically pinned to CPU cores for predictable performance. The application manages packet buffers and assigns queues to processing cores (DPDK Project, n.d.).

netmap reduces packet I/O overhead through preallocated buffers shared between kernel and user space and batched packet I/O (Rizzo, 2012). It provides efficient packet I/O for applications such as user-space Click.

The following subsection examines eBPF and XDP as in-kernel approaches to programmable packet processing (Høiland-Jørgensen *et al.*, 2018).

2.4.3 eBPF

eBPF provides programmability inside the Linux kernel. For network acceleration, XDP allows verified eBPF programs to execute in network-driver context before the conventional networking stack. Early packet dropping or redirection can reduce subsequent stack-processing overhead, supporting functions such as packet filtering and load balancing (Høiland-Jørgensen *et al.*, 2018)[3].

[3] For example, Cloudflare uses eBPF for programmable packet filtering in Magic Firewall (Arges, 2021) and XDP for packet forwarding in its Unimog load balancer (Wragg, 2020).

Origins and Evolution eBPF traces its origins to the *Berkeley Packet Filter* (BPF), a lightweight bytecode mechanism introduced by McCanne and Jacobson (1993) for kernel-based filtering of captured packets delivered to user-space applications.

Decades later, the original BPF design was substantially extended to produce eBPF (Høiland-Jørgensen *et al.*, 2018). The key advances included an in-kernel verifier that enforces safety properties on bytecode prior to execution, a *Just-in-Time* (JIT) compiler that translates verified bytecode to native machine code, and a richer set of kernel hook points to which eBPF programs can be attached. eBPF programs are event-driven, triggered by specific kernel or user-space events, such as a network packet arriving at a NIC, a function being called, or a tracepoint firing. At runtime, eBPF programs may invoke built-in helper functions to interact with kernel subsystems and exchange data with user space.

Today, eBPF has evolved into a general-purpose in-kernel runtime that supports a wide range of applications beyond its original packet filtering use

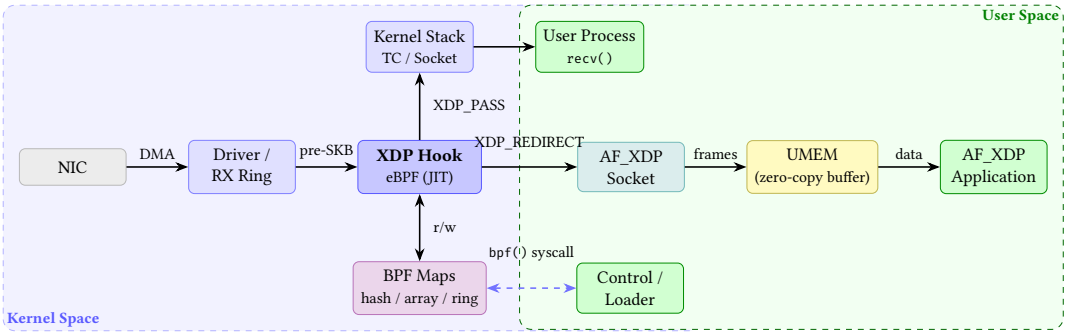


Figure 2.3: XDP packet-processing pipeline.

case. Its network processing capabilities in particular remain a core focus, and eBPF programs can be attached at various hook points in the kernel networking stack, including XDP (at the driver layer, before the kernel networking stack), *Traffic Control* (TC) (traffic control layer), socket filters, kprobes (arbitrary kernel functions), and tracepoints (stable instrumentation points). BPF bytecode can additionally be executed by user-space interpreters such as uBPF, which bypass the kernel verifier and support more complex logic at the cost of the performance and isolation guarantees of in-kernel execution (Holik, Jouet & Pezaros, 2025). As illustrated in fig. 2.3, the XDP hook exposes two principal packet paths: a fast path that redirects frames directly to an AF_XDP socket and onwards to user space via a shared UMEM buffer, bypassing the kernel networking stack, and a slow path that passes frames to the conventional TC and socket layers; BPF Maps serve as the shared state store accessible from both the eBPF program and user space via the `bpf()` system call.

Data Sharing Mechanisms The central data-sharing mechanism introduced by eBPF is BPF Maps: typed key-value stores allocated in kernel memory and accessible by both eBPF programs and user-space applications via the `bpf()` system call. BPF Maps support a wide variety of typed structures, including hash maps, arrays, *Least Recently Used* (LRU) caches, per-CPU variants, and ring buffers. Map capacity is fixed at creation time and bounded only by system memory[4], which decouples stateful storage from the on-chip SRAM constraints of specialised switching hardware. Ring buffer maps further support asynchronous, low-overhead event notification from the kernel to user space, enabling real-time telemetry and security monitoring without per-packet signalling overhead (Holik, Cook *et al.*, 2025; Vieira *et al.*, 2020).

[4] More precisely, map memory is subject to `RLIMIT_MEMLOCK` in older kernels or cgroup memory quotas in newer ones; however, these limits are considerably larger than the on-chip SRAM or TCAM available in specialised switching hardware.

Toolchains and Frameworks A number of frameworks have been developed to facilitate eBPF programming, serving as scaffolding layers that abstract low-level details such as kernel compatibility, system calls, and program loading. While these frameworks, such as BCC (yonghong-song *et al.*, 2025), ebpf-go (L. Bauer *et al.*, 2025), and aya-rs (Duberstein *et al.*, 2025), enhance the programming experience and streamline the development process, they

do not alter the final artefact produced by the compiler, nor do they affect the behaviour of the eBPF verifier and the in-kernel virtual machine.

Regardless of the framework employed, all eBPF programs must pass through a fixed compilation pipeline (Vieira *et al.*, 2020): they are first written in a restricted subset of C and compiled by Clang/LLVM into architecture-independent eBPF bytecode, which is then submitted to the kernel via the `bpf()` system call for safety verification. If the program is accepted by the verifier, a JIT compiler translates the bytecode into native machine code for execution; otherwise, the program is rejected and cannot be attached to any hook point.

Constraints and Limitations While the verifier enforces program safety, it also imposes constraints that fundamentally shape what eBPF programs can express (Vieira *et al.*, 2020). The control flow graph of every eBPF program must form a *Directed Acyclic Graph* (DAG); programs containing unbounded loops are rejected outright, although recent Linux kernels introduced support for bounded loops whose iteration count can be statically proven finite. Each program is further subject to a hard 512-byte stack limit and, by default, a verifier complexity limit; programs that exceed these bounds must have their logic decomposed across multiple programs linked via tail calls (up to 33 levels as mandated by the Linux kernel) or BPF-to-BPF function calls (Cilium, 2026). Moreover, every pointer dereference must be statically proven safe by the verifier, and access to BPF Map values requires an explicit null check prior to dereferencing.

2.4.4 Comparative Analysis

P4 and eBPF represent two fundamentally distinct programmability paths: the former targets line-rate hardware pipelines on merchant silicon, the latter retrofits programmability into the Linux kernel on commodity hosts. They are complementary technologies whose appropriate choice is governed by deployment context, state requirements, and performance objectives (Hauser *et al.*, 2023; Michel *et al.*, 2021). The remainder of this section compares them across five dimensions (execution model, deployment targets, state management and control-plane interaction, programming constraints, and performance versus flexibility) and summarises their contrasts and complementarities in table 2.1.

Execution Model P4 operates on a match-action pipeline model, in which each packet traverses a sequence of stages in order; multiple packets may occupy different stages simultaneously, giving the pipeline its throughput efficiency. The per-packet processing latency is deterministic, bounded by the stage count and per-stage delay defined in the target architecture (Bosshart *et al.*, 2014). Unlike P4, eBPF is event-driven and hook-based: a program

is triggered upon a specific kernel or user-space event, such as a packet arriving at a NIC. Execution latency therefore depends on the hook point; hooks at the TC or socket-filter layer are subject to kernel scheduler and interrupt-handling overhead, whereas XDP programs, which execute at the driver layer before the networking stack, achieve more deterministic latency (Høiland-Jørgensen *et al.*, 2018; Vieira *et al.*, 2020). In general, P4 guarantees predictable per-packet processing time at the cost of general-purpose expressiveness, whilst eBPF offers greater flexibility at the cost of latency determinism.

Deployment Targets P4 targets a range of hardware platforms, with its execution model defined by the underlying target architecture. Its primary deployment environment is the programmable ASIC, of which the Intel Tofino family is the most widely adopted in both industry and academic research (Agrawal & Kim, 2020; Hauser *et al.*, 2023). Beyond dedicated switching ASICs, P4 is also supported on FPGA-based platforms via vendor-specific toolchains such as AMD Vitis Networking P4 for the Alveo accelerator family, which are used in research and network function virtualisation contexts. For educational and prototyping purposes, software targets such as the behavioural model (BMv2) and the P4Pi project for Raspberry Pi (Laki *et al.*, 2021) provide accessible alternatives without specialised hardware.

Conversely, eBPF targets commodity Linux hosts and requires no specialised hardware; it runs on modern Linux servers with a sufficiently recent Linux kernel. As a core component of the Linux kernel, it is supported across a wide range of Linux distributions. Early SmartNICs such as the Netronome Agilio family supported native eBPF offload, enabling eBPF programs to execute directly on the NIC hardware and bypass the host CPU for latency-sensitive workloads[5] (Kicinski & Viljoen, 2016).

[5] Netronome ceased active development of the Agilio *Netronome Flow Processor* (NFP) product line, so native eBPF NIC offload is no longer supplied by mainstream commercial SmartNIC vendors as of the time of writing.

State Management and Control-Plane Interaction P4 and eBPF differ substantially in how data-plane programs maintain state and exchange information with the control plane.

In P4, the principal mechanisms include the following: register arrays provide fast, SRAM-backed storage accessible via the control-plane API, but their capacity is constrained by the on-chip memory budget of the target device; packet replication or forwarding through a dedicated CPU port allows the control plane to inspect full headers and payloads, but incurs per-packet processing overhead; and Digest messages convey custom metadata to the control plane through a P4Runtime gRPC stream, with payload structure fixed by the P4 program at compile time and unidirectional semantics. Counters, meters, mirror sessions, and architecture-specific asynchronous calls (e.g. in *Portable NIC Architecture* (PNA)) provide further channels but are typically used for narrower purposes. Collectively, these mechanisms are tightly

coupled to the hardware pipeline and offer limited capacity for large or dynamically structured state.

eBPF, by contrast, exposes state through BPF Maps (section 2.4.3), which afford bidirectional, runtime data exchange via the `bpf()` system call and asynchronous notification through ring buffers, all without packet-level signalling. This architectural difference makes eBPF substantially better suited for network functions that require large, structured, or rapidly evolving state, whereas P4's tighter hardware integration makes it the preferable choice where deterministic line-rate forwarding is the primary objective (Holik, Cook *et al.*, 2025).

Programming Constraints According to the P4 specification, P4 programs do not support general-purpose loops in primitive language constructs. Two workarounds exist: the FSM model of the parser allows state transitions that can simulate bounded iteration over header stacks, and packet recirculation enables a packet to re-enter the pipeline for additional processing passes (Akem *et al.*, 2023). eBPF achieves expressiveness via a different route: subject to the verifier-mandated DAG control flow detailed in section 2.4.3, it relies on bounded loops, BPF-to-BPF function calls, and tail calls instead of P4's parser-FSM and recirculation paradigms.

P4's programming constraints are primarily driven by the need to guarantee deterministic, line-rate processing on hardware pipelines, whereas eBPF's are designed to ensure safety and verifiability within the Linux kernel; for workloads that require greater expressiveness, user-space BPF interpreters such as uBPF and rbpf provide alternative execution environments that are not subject to the kernel verifier, at the cost of trading off performance and safety guarantees (Holik, Jouet & Pezaros, 2025).

Performance and Flexibility Direct performance comparison between P4 and eBPF is inherently target- and workload-dependent; nevertheless, characteristic performance properties can be drawn from the respective design philosophies.

Once a P4 program is compiled and deployed on a hardware target, typically a switch built on programmable ASICs, the match-action pipeline can forward traffic at line rate up to the Tbps scale (Agrawal & Kim, 2020), with per-packet processing latency in the sub-microsecond range dominated by the pipeline traversal time. However, P4 programs remain subject to the hardware resource limits detailed in section 2.4.1, which constrain expressiveness unless offloaded through `extern` objects and tie the program closely to the underlying hardware architecture. eBPF/XDP, as its name (eXpress Data Path) suggests, is also designed for low-latency packet processing and can achieve sub-microsecond per-packet latency in optimised configurations on commodity hardware (Høiland-Jørgensen *et al.*, 2018). However, it is sub-

ject to CPU scheduling and interrupt-handling overhead, which makes its latency less deterministic than that of an ASIC. In exchange, eBPF programs can be replaced at runtime with minimal disruption to ongoing traffic, offering an operational agility that hardware-compiled P4 targets cannot match.

Table 2.1: Comparison of P4 and eBPF across five dimensions.

Dimension	P4	eBPF
Execution model	Match-action pipeline; sequential per-stage execution with packet-level parallelism across stages; deterministic latency bounded by pipeline stage count	Event-driven hook triggered per kernel/user-space event; TC and socket-filter latency subject to scheduler and interrupt overhead, whereas XDP at the driver layer achieves more deterministic latency
Deployment target	Programmable ASIC (Intel Tofino); FPGA accelerator (AMD Alveo via Vitis Networking P4); software targets (BMv2, P4Pi on Raspberry Pi)	Commodity Linux x86/ARM hosts; SmartNICs with native eBPF offload (e.g. Netronome Agilio)
State management and control-plane interaction	Register arrays (SRAM-backed, bounded by on-chip memory budget) via Control-Plane API; CPU-port replication for full-packet inspection at per-packet cost; Digest messages for lightweight metadata, unidirectional only	BPF Maps (hash, array, LRU, per-CPU) fixed at creation, bounded by system memory; `bpf()` system call for bidirectional, runtime data exchange; ring buffer maps for low-overhead asynchronous notifications
Programming constraints	No general-purpose loops; parser FSM simulates bounded iteration over header stacks; packet recirculation enables additional processing passes	DAG control flow enforced by verifier; bounded loops with statically provable termination; BPF-to-BPF function calls and tail calls (up to 33 levels) for structuring complex logic
Performance and flexibility	Sub-µs per-packet latency; Tbps line rate (e.g. Tofino 2 up to 12.9 Tbps (Agrawal & Kim, 2020)); expressiveness constrained by PHV count and pipeline stage budget; tied to hardware after compilation	Sub-µs per-packet latency on XDP; less deterministic due to CPU scheduling and interrupt overhead; atomic runtime replacement without traffic disruption

Taken together, these dimensions reveal a division of labour: P4 is well matched to bounded, data-plane-maintained workloads, whereas eBPF supports host-resident processing with runtime state shared with user space. Section 2.5 formalises the corresponding differences in state ownership and data-plane independence.

2.5 A State-Ownership Framework for In-Network Offloading

Building on the complementarity between P4 and eBPF identified in section 2.4.4, this section organises in-network offloading by state ownership and data-plane independence, as defined in Section 2.2.

State ownership may reside entirely in the data plane or be shared with control-plane or user-space software. Persistent state and cross-packet updates are compatible with data-plane ownership when the data plane performs the required runtime maintenance.

Ownership and independence are distinct: shared state ownership can coexist with an independent fast path. The classifications below concern normal runtime operation; deployment, passive readout, and exceptional reconfiguration are assessed separately.

2.5.1 Data-Plane State Ownership

In this arrangement, the data plane maintains the function's runtime state and completes its per-packet decisions using packet content and that state. The required operations must fit within the target's processing and memory limits, potentially using recirculation for multiple pipeline passes. These limits constrain the supported state representation and service behaviour.

Chapter 3 investigates these costs and restrictions through a *Domain Name System* (DNS) cache.

2.5.2 Shared State Ownership

In this arrangement, control-plane or user-space software participates in maintaining the function's runtime state or policy. Such participation may be needed when state or processing exceeds the chosen data plane's capabilities, as in some connection-tracking, anomaly-detection, or SYN-flood protection designs (Teng *et al.*, 2022).

Coordination may use a control API such as P4Runtime, a dedicated CPU port that redirects selected packets to software, or a shared-memory interface such as BPF Maps. Software can maintain policy or other shared state in response to cache misses (Woodruff *et al.*, 2019), threshold crossings, or periodic timers.

When a packet requires a synchronous software decision, that interaction adds latency and reduces processing determinism. Asynchronous updates instead affect when revised state becomes visible to subsequent packets,

without making the fast path wait for software. The fast path remains subject to the processing and parsing bounds of its target (Chen *et al.*, 2024; Franco *et al.*, 2024; Scazzariello *et al.*, 2023).

Chapter 4 investigates this arrangement through a *Generic Object Oriented Substation Event* (GOOSE) detector whose in-kernel sequence checks operate independently while user space aggregates events and updates enforcement policy.

2.6 Application Domains and Research Gaps

Whereas section 2.5 organised in-network offloading along the state-ownership dimension, this section revisits the same body of work along an orthogonal axis, namely *application domain*. Each domain includes different ownership arrangements, so reading the two views together illustrates their maintenance responsibilities and processing dependencies. The review selects representative works on network services and traffic inspection relevant to this thesis.

Concretely, the first subsection covers in-network caching and protocol-specific services. The surveyed systems use different cache-management paths, even when cache-hit lookups remain independent. The costs of maintaining a protocol-specific cache entirely within the data plane motivate chapter 3. The second subsection shifts the emphasis from service-traffic acceleration to traffic inspection, flow-level anomaly detection, and in-network machine-learning inference, a setting that includes data-plane-maintained statistics and inference as well as eBPF detectors with software-maintained policy, first in *Information Technology* (IT) traffic and, more recently, in *Operational Technology* (OT) protocols; this trajectory motivates the shared-ownership architecture developed in chapter 4.

Table 2.2 summarises the state ownership and data-plane independence of the reviewed implementations.

2.6.1 In-Network Caching, Key-Value Stores, and Protocol-Specific Services

In-network caching is among the most direct applications of in-network offloading: a transparent middlebox placed on the data path serves hot items without reaching backend servers. This approach is particularly effective for workloads with a skewed access pattern, especially key-value stores such as memcached and DNS, where a small fraction of heavy hitters account for a large portion of the traffic.

Cache lookups can be served using per-packet keys matched against data-plane-resident objects. The systems differ in who maintains cached items:

some use a control-plane process, while others perform updates entirely on the data path. Cache-hit processing can remain independent in either case.

An early proposal for in-network caching was NetCache (X. Jin *et al.*, 2017), which placed a key-value cache on a programmable switch using match-action tables for lookups and register arrays for value storage. Although per-packet lookups are served entirely in the data plane, the cache management is implemented by the control plane: the switch tracks per-entry access for cached items in registers and screens uncached keys with a Count-Min Sketch heavy-hitter detector, and the control plane periodically promotes or evicts keys on this evidence. NetCache demonstrated that this arrangement could serve rack-scale workloads at line rate, but is limited to a single switch and does not address scale-out via cache pooling across multiple switches.

Subsequent work has mitigated the single-switch bottleneck by replicating the cache across multiple switches, but the resulting systems assign state-maintenance responsibility differently. NetChain (X. Jin *et al.*, 2018) implemented chain replication entirely within the data plane to provide strong consistency guarantees: reads, writes and inter-switch coordination all proceed without invoking user space on the per-packet path, with the controller consulted only for failure recovery and chain reconfiguration.

NetCRAQ (Sagkriotis & Pezaros, 2022) further refined the underlying replication algorithm by applying Chain Replication with Apportioned Queries (CRAQ (Terrace & Freedman, 2009)) to the NetChain setting, reducing read latency by serving reads from any replica after a version check while preserving linearisability.

DistCache (Z. Liu *et al.*, 2019) distributes the cache across two switch tiers (spine and leaf) using independent hash functions, and at lookup time routes each query to the less-loaded of its two candidate cache switches by power-of-two-choices; cache *placement*, however, is computed by a controller that sits off the per-query critical path, while each switch refreshes the objects within its own partition through decentralised replacement managed by switch-local software agents. Pegasus (J. Li *et al.*, 2020) extends static hash-partitioning with selective replication of hot keys, maintaining an in-network coherence directory on the switch that rebalances each object's replica set on every write, with a software controller selecting hot keys for replication to counter workload skew. Both systems use software for cache management while keeping per-packet dispatch in the data plane.

A related line of work addresses storage capacity. TEA (D. Kim *et al.*, 2020) proposed a virtual flow-table abstraction for stateful network functions (*Network Address Translation* (NAT), stateful firewall, load balancer) that externalises large lookup tables to server-side *Dynamic Random-Access Memory* (DRAM) accessed by the switch via in-data-plane *Remote Direct Memory Access* (RDMA), retaining only a small cache on-chip for hot entries. The control plane inserts and deletes table entries and maintains the on-chip cache,

while lookups proceed through data-plane RDMA.

More recently, P4LRU (Zhao *et al.*, 2023) implemented an LRU eviction policy within the data plane, demonstrating that even with limited on-chip memory, intelligent eviction encoded entirely in data-plane tables and registers can significantly improve hit rates.

Among protocol-specific applications, the DNS protocol is a natural fit for in-network caching due to its read-heavy workload and highly skewed traffic pattern. P4DNS (Woodruff *et al.*, 2019) implemented a DNS cache in the data plane, with software resolving recursive misses and maintaining the cache. A separate type of P4-based DNS work targets security policy enforcement. P4DDPI (AlSabeh *et al.*, 2022) and DPIDNS (Ahad *et al.*, 2023) learn domain-to-*Internet Protocol* (IP) mappings from DNS replies and use them to filter subsequent traffic in the data plane. DPIDNS uses recirculation to parse additional labels, but delegates labels longer than 19 bytes to the controller. HELP4DNS (Şahin & Demirci, 2025) addresses a different threat, DNS flood and amplification, through concurrent-query limiting and DNS-response validation, including fragmented responses, on the data path.

The surveyed systems divide cache-maintenance responsibilities differently. NetCache, DistCache, Pegasus and P4DNS use software to maintain cache contents or handle misses while keeping cache-hit processing independent. NetChain's normal-case chain replication and P4LRU's data-plane LRU eviction keep state updates on the data path. The surveyed management primitives and replication systems maintain state on the data path, while protocol-specific caching introduces additional parsing, invalidation, and refresh responsibilities. Chapter 3 addresses this gap with a P4-based DNS cache that runs entirely in the data plane and evaluates this arrangement end-to-end, characterising the resource costs and service trade-offs required for data-plane independence.

2.6.2 Traffic Monitoring, Anomaly Detection, and In-Network Intelligence

Traffic inspection work spans in-network measurement, anomaly detection, and lightweight machine-learning inference. These systems range from data-plane-maintained telemetry and inference to detectors with software-coordinated analysis or policy updates. The subsection therefore proceeds from data-plane-maintained mechanisms to mechanisms with software coordination.

PINT (Ben Basat *et al.*, 2020) encodes telemetry in fixed-size packet digests that switches update independently; a recording module stores the extracted digests, and software reconstructs the requested information. P4NetIntel (Puttlitz *et al.*, 2024) combines network and host telemetry across request and response paths. P4 switches insert and extend *In-band Network Tele-*

metry (INT) headers; a server-side eBPF XDP program stores the extracted telemetry and host metrics in kernel maps indexed by *Remote Procedure Call* (RPC) request ID, and removes the INT headers before application delivery. A TC egress program retrieves the stored information and appends it to the response; the INT sink exports the telemetry and removes the headers. Its prototype includes BMv2 and NVIDIA BlueField-2 DPUs targets.

For switch-local statistics, sketch-based approaches are widely used to detect heavy flows and estimate traffic statistics (Ding *et al.*, 2022). SketchLib (Namkung *et al.*, 2022) provides an optimisation framework for deploying such sketches efficiently on programmable switches, while Choueiri *et al.* (2024) extend a Count-Min Sketch-based heavy-hitter detector for *Denial of Service* (DoS) monitoring to a P4-DPDK software substrate. Building on this family, Harmonia (W. Li, Chu *et al.*, 2025) targets heavy-flow detection with improved accuracy under tight switch memory budgets, while Pallas (W. Li, Bütün *et al.*, 2025) shifts the focus to *persistent*-flow detection and removes any control-plane dependency from the detection path. These approaches maintain measurement counters on the packet-processing path. SketchLib additionally uses controller-installed entries for heavy-flow-key tracking. In parallel co-authored work, Charon (W. Li *et al.*, 2026) adds priority awareness to this family through a stratified-sampling admission stage that admits packets in proportion to their marked priority rank (*Differentiated Services Code Point* (DSCP) field), concentrating measurement accuracy on high-priority flows as the memory budget tightens; its primary realisation is a CPU-resident software sketch, with a validated Tofino prototype.

For switch-local intelligence, recent work has explored the deployment of machine learning inference models within the data plane (Zheng *et al.*, 2023) to enable real-time, in-network decision-making for complex tasks such as anomaly detection and traffic classification. For example, NetPixel (Siddique *et al.*, 2021) accumulates image features across image-chunk packets and applies a lightweight decision tree in the data plane. Karkashina *et al.* (2024) then proposed hierarchical decision-tree inference for flow-level anomaly detection, maintaining flow features and classification state in switch registers. Building on this, the original IIsy (Xiong & Zilberman, 2019) mapped decision trees, support vector machines, naive Bayes and K-means classifiers onto P4 pipelines; a subsequent work (Zheng *et al.*, 2024) extends model support and combines a small switch model with a larger backend model, sending low-confidence inputs to the backend for inference. The trained models are configured before inference; NetPixel and flow-level inference additionally maintain features across packets.

To extend packet-processing functionality at runtime, *Honey for the Ice Bear* (Simon *et al.*, 2024) embeds eBPF processing within the T4P4S software P4 target and supports runtime code updates. Its eBPF components execute as part of the software data plane.

Other approaches optimise software packet processing or distribute pro-

grams across multiple targets. Morpheus (Miano *et al.*, 2022) specialises eBPF and DPDK packet-processing code at runtime using control-plane configuration and observed traffic patterns. Flightplan (Sultana *et al.*, 2021) splits P4 programs into cooperating components placed across heterogeneous data-plane targets, including switches, FPGAs, and servers, to exploit their different capabilities while preserving program behaviour.

A concrete security application is SmartCookie (Yoo *et al.*, 2024), which combines the strengths of P4 and eBPF through a split-proxy defence against SYN floods. The P4 switch performs cryptographic SYN-cookie checks and maintains approximate connection records, while eBPF programs on the server maintain exact connection state and translate *Transmission Control Protocol* (TCP) sequence numbers. This division combines high-speed filtering in the switch with the server's greater capacity for per-connection state.

Beyond SYN-flood mitigation, eBPF also supports traffic inspection through coordination between kernel and user-space components. Kernel programs can maintain per-flow state in BPF maps, while user-space analysis updates the policy used by the packet-processing path. Related work has extended eBPF-based processing to OT protocols such as IEC 61850 GOOSE (Holik, Jouet & Pezaros, 2025). A further challenge is to incorporate physical-process context into anomaly assessment while retaining an independent packet-processing path. Chapter 4 investigates this through in-kernel GOOSE checks, user-space policy maintenance, and a co-simulation interface.

Table 2.2: Implementations mapped to the state-ownership framework.

Implementation	State ownership	Data-plane independence
NetCache (X. Jin *et al.*, 2017)	Shared	Independent cache-hit path; software maintains cache contents.
NetChain (X. Jin *et al.*, 2018)	Data plane	Independent reads, writes, and replication in normal operation.
NetCRAQ (Sagkriotis & Pezaros, 2022)	Data plane	Independent read and replication paths in normal operation.
DistCache (Z. Liu *et al.*, 2019)	Shared	Independent query dispatch; switch-local software agents maintain cache contents.
Pegasus (J. Li *et al.*, 2020)	Shared	Independent packet dispatch; software participates in replica management.
TEA (D. Kim *et al.*, 2020)	Shared	Independent table lookups via RDMA; software maintains tables and the on-chip cache.
P4LRU replacement primitive (Zhao *et al.*, 2023)	Data plane	Independent replacement-state updates and eviction.
P4DNS (Woodruff *et al.*, 2019)	Shared	Independent cache hits; software resolves recursive misses and maintains the cache.
P4DDPI (AlSabeh *et al.*, 2022)	Data plane	Independent filtering for supported domains; domain-to-IP state is learned in the data plane.
DPIDNS (Ahad *et al.*, 2023)	Data plane (learned mappings)	Independent learning and filtering for supported labels; labels longer than 19 bytes require controller processing.
HELP4DNS (Şahin & Demirci, 2025)	Data plane	Independent concurrent-query limiting and DNS-response validation.
SmartCookie (Yoo *et al.*, 2024)	Shared (key rotation); connection state in data planes	Packet checks avoid synchronous controller decisions; connection setup coordinates switch and kernel agents.
PINT (Ben Basat *et al.*, 2020)	Software (recorded telemetry)	Independent packet encoding; software records digests and reconstructs telemetry off path.

Continued on next page

Table 2.2 (continued)

Implementation	State ownership	Data-plane independence
P4NetIntel (Puttlitz *et al.*, 2024)	Data plane (kernel correlation maps)	Independent telemetry processing in switches and kernel hooks; XDP stores request data and TC retrieves it for responses.
SketchLib (Namkung *et al.*, 2022)	Shared (sketch and flow-key tracking)	Independent sketch updates; the controller maintains auxiliary flow-key entries.
P4-DPDK DoS monitoring (Choueiri *et al.*, 2024)	Data plane (measurement state)	Independent sketch updates on the software data plane.
Harmonia (W. Li, Chu *et al.*, 2025)	Data plane	Independent heavy-flow measurement.
Pallas (W. Li, Bütün *et al.*, 2025)	Data plane	Independent persistent-flow detection.
Charon (W. Li *et al.*, 2026)	Data plane (sampling and sketch state)	Independent priority admission and sketch updates on CPU and Tofino targets.
NetPixel (Siddique *et al.*, 2021)	Data plane (image-feature state; fixed model)	Independent feature accumulation and image classification after model deployment.
Hierarchical decision-tree inference (Karkashina *et al.*, 2024)	Data plane (flow features and classification state)	Independent flow-feature updates and inference with the configured model.
IIsy (Xiong & Zilberman, 2019)	No mutable model state during inference	Independent inference for the configured model.
Extended IIsy toolchain (Zheng *et al.*, 2024)	No mutable model state during inference	Independent high-confidence classification; low-confidence inputs require backend inference.
BPFabric OT resilience functions (Holik, Jouet & Pezaros, 2025)	Data plane (counters and fixed policy)	Configured functions process packets independently; the controller orchestrates deployment.
Honey for the Ice Bear (Simon *et al.*, 2024)	Function-dependent	Installed P4/eBPF functions execute within the software data plane.
Morpheus (Miano *et al.*, 2022)	Function-dependent	Packets use specialised code or fall back to the original datapath during runtime optimisation.

Continued on next page

Table 2.2 (continued)

Implementation	State ownership	Data-plane independence
Flightplan (Sultana *et al.*, 2021)	Function-dependent	Configured data-plane segments cooperate without per-packet controller decisions.
Register-based DNS cache in chapter 3	Data plane	Independent cache operations within the supported service bounds.
GOOSE detector in chapter 4	Shared	Independent kernel checks; user-space event analysis and asynchronous policy updates.

For the inference entries, the model parameters are configured before execution and require no runtime maintenance during inference. For Honey for the Ice Bear, Morpheus, and Flightplan, state ownership depends on the deployed function; the execution framework alone does not determine who creates, updates, or invalidates that function's runtime state.

2.7 Summary

This chapter has examined the two prevailing programmable data plane technologies, P4 and eBPF, and characterised their respective capabilities, as summarised in table 2.1. P4 delivers deterministic, line-rate processing on programmable ASICs, but its fixed stage budget, on-chip SRAM ceiling, and absence of native general-purpose loops constrain its capacity to maintain rich cross-packet state. eBPF, by contrast, trades off a measure of latency determinism in exchange for scalable stateful storage in user space and bi-directional communication. Their complementarity, grounded in contrasting deployment targets and state models, underpins the framework in section 2.5.

Building on this complementarity, the chapter set out an analytical framework organised by state ownership and data-plane independence and used it to examine the related work surveyed in section 2.6. This review establishes the background and motivation for the subsequent chapters. The DNS cache and GOOSE detector then provide evidence of the resource costs, service restrictions, and coordination requirements of their respective ownership arrangements; their synthesis in Chapter 5 develops the thesis-level guidance on offloading decisions.

The survey showed how data-plane-owned management primitives and replication mechanisms maintain runtime state independently, while protocol-specific services add parsing and cache-lifecycle obligations. Realising the caching path of a real protocol such as DNS entirely within the data plane,

with no control-plane slow path, therefore exposes the resource and processing costs, and the restrictions on cache behaviour, required for independent data-plane operation.

Related work has extended eBPF-based processing to OT protocols such as IEC 61850 GOOSE (Holik, Jouet & Pezaros, 2025). This thesis develops a GOOSE analysis function that partitions protocol checks and policy maintenance between an independent kernel fast path and user space. The prototype exposes the coordination requirements and reaction costs of asynchronous policy updates based on packet-derived anomaly events. Integration with a digital-twin co-simulation is presented as a proposed use case that supplies physical-process context for future anomaly assessment. This context interface extends the available information sources; the evaluated detection and enforcement paths use protocol-derived state.

The remainder of the thesis addresses these gaps in turn. Chapter 3 fills the first gap by presenting a P4-based DNS cache and using it to characterise the resource and processing costs, and the service trade-offs, of maintaining cache state entirely within the data plane. Chapter 4 presents an architecture and prototype for OT detection with kernel checks and user-space policy maintenance; the chapter develops the architectural proposal, implements the in-kernel detection pipeline, and outlines what remains before production deployment. Chapter 5 then presents guidelines for stateless and stateful applications based on lessons from both systems, relating state ownership and data-plane independence to the selection of an offloading arrangement.

Chapter 3

DNS Caching with Data-Plane State Ownership

3.1 Overview

This chapter examines the resource and processing costs, and the restrictions on service behaviour, of a *Domain Name System* (DNS) cache that maintains its runtime state and completes per-packet decisions entirely within the data plane. Building on the state ownership and data-plane independence framework introduced in Chapter 2, it addresses RQ1 through a P4 implementation evaluated on the *Behavioral Model v2* (BMv2) software target. This implementation provides evidence for the first thesis claim (*so*, Section 1.4): independent state maintenance and packet processing remain subject to resource and service constraints.[1]

DNS caching provides a service in which independent packet processing also requires the data plane to maintain cached records over time. The prototype targets cleartext DNS over *User Datagram Protocol* (UDP), with each supported query and response contained within a single packet, and assumes IPv4 A-record lookups (Mockapetris, 1987b).

The remainder of this chapter is organised as follows. Section 3.2 introduces the DNS protocol, and Section 3.3 motivates in-network DNS caching. Section 3.4 presents the system architecture and implementation, followed by the analysis of design choices and trade-offs in Section 3.5. Section 3.6 evaluates resource usage, cache hits, per-packet processing, and cache-hit latency against a match-action baseline; Section 3.7 draws lessons for P4 function development, discusses limitations, and summarises the findings.

[1] This chapter builds on the published system of X. Li and Pezaros (2025). The thesis version adds epoch-based invalidation, replaces three-way probing with two-slot probing and a verification fingerprint, aligns the register-array size with the hash modulus, and uses binary decomposition for padding removal. It also includes a consolidated design rationale and comparison with a re-implemented match-action baseline.

3.2 Background

DNS (Mockapetris, 1987a; 1987b) maps domain names to *Internet Protocol* (IP) addresses through a hierarchical, distributed namespace, and operates as a key–value store that underpins the addressing layer of essentially every Internet transaction. Two server roles are conventionally distinguished. An *authoritative server* returns answers drawn from the zone it administers; a *recursive resolver* answers a client query on its behalf by traversing the delegation hierarchy from the root through the relevant top-level and authoritative servers, caching intermediate results along the way. The cache examined in this chapter is positioned at, or in front of, a recursive resolver, where repeated lookups concentrate and where reduced lookup latency translates directly into improvements perceived by clients.

A DNS message consists of a fixed 12-byte header followed by a small number of variable-length sections: a question section that names the resource being queried, and answer, authority, and additional sections that carry resource records returned by a server. The QNAME field in the question section is encoded as a sequence of length-prefixed labels terminated by a zero octet, so that a name such as `google.com.` occupies twelve bytes structured as `[6]google[3]com[0]`. Each resource record carries a type, a class, a *Time-to-Live* (TTL) value that bounds the cache lifetime of the record at any resolver, and a variable-width data field whose contents are determined by the type. The most common record types are A for IPv4 addresses, AAAA for IPv6 addresses, and CNAME for canonical-name aliases.

RFC 1035 specifies a 512-byte message limit for DNS over UDP (Mockapetris, 1987b). DNS over *Transmission Control Protocol* (TCP) lifts the per-packet bound at the cost of connection state; *DNS over TLS* (DoT) and *DNS over HTTPS* (DoH) additionally impose *Transport Layer Security* (TLS) session state. The work in this chapter targets cleartext DNS over UDP, with supported A-record queries and responses fitting within this limit and a single packet, as required by the cache design.

3.3 Motivation

The general case for caching read-heavy, popularity-skewed workloads on programmable switches has been made in Section 2.6. What needs to be emphasised in the DNS setting is the empirical concentration of queries onto a small number of provider categories: hyperscaler *Application Programming Interface* (API) endpoints, image *Content Delivery Networks* (CDNs), and social-media CDNs (exemplified, as of 2024, by domains such as `googleapis.com`, `gstatic.com`, `amazonaws.com` and `fbcdn.net`) consistently dominate domain popularity rankings, according to Cloudflare Radar (Cloudflare, 2026), Cisco Umbrella ('Cisco Popularity List', 2016) and Tranco (Pochat *et al.*,

2018). The recent measurement work of Dai *et al.* (2025) further substantiates this observation by analysing a large corpus of DNS traces and showing this skew to be a general property of DNS traffic.

A second, DNS-specific observation concerns the high availability inherent in the protocol's deployment, which directly relaxes the consistency requirements an in-network cache must meet. For large network service providers, a domain name is typically associated with multiple A or AAAA records, and public resolvers return only a subset of these, frequently rotated or selected geographically. Public DNS providers often apply anycast routing, which further distributes traffic by advertising the same IP address resource from multiple geographically dispersed data centres, as exemplified by the deployments behind Google's `8.8.8.8` and Cloudflare's `1.1.1.1`. The practical consequence is that even when a client receives, and an intermediate cache holds, only a single record for a service, the path to that service remains resilient to individual resolver and server failures: a single record typically resolves to an anycast endpoint or a load-balanced virtual IP representing multiple servers, and application-layer timeouts will in any case trigger fresh resolutions should the cached path fail. A modest staleness window or a missing alternative record is therefore unlikely to compromise reachability, which in turn legitimises a caching discipline that trades strict consistency with the authoritative source for fully data-plane-resident operation [2].

Taken together, popularity skew and protocol-level redundancy motivate three operational properties on which the cache design rests. First, the working set is *compressible*: the overwhelming majority of traffic concentrates onto a tractable subset of entries that fits within an on-chip footprint of a few bytes per entry. Second, lookups are *loss-tolerant*: a cache miss degrades latency but never compromises correctness, since the authoritative path remains reachable upstream and a cache need not be perfect to be beneficial. Third, records are *truncatable*: bounded-width fixed-size slots impose a deliberate lossy compression of the long tail, given that the heavy tail contributes negligibly to query volume. These three properties recur across the design decisions analysed in Section 3.5.

The update path is central to realising these caching properties. The most directly comparable prior system, P4DNS (Woodruff *et al.*, 2019), mirrors DNS responses to a runtime control plane via *Direct Memory Access* (DMA); the control plane in turn writes the cache entries into the data-plane match-action tables. That coupling, however, makes every cache write depend on the latency and reliability of the control-plane channel.

The present chapter departs from this arrangement on three fronts: runtime control-plane involvement is eliminated entirely from the per-packet path;

[2] This relaxation aligns with the *eventual consistency* already inherent in DNS itself, where authoritative updates propagate through recursive resolvers as TTL-bounded soft state.

the cache is kept current through a passive, probabilistic refresh mechanism whose precise form is justified in Section 3.5.4; and feasibility is established on the BMv2 software P4 target, with portability to *Application-Specific Integrated Circuit* (ASIC) targets reserved for future work and discussed as a limitation in Section 3.7.2. Together, these choices enable an end-to-end assessment of the costs of maintaining the cache entirely within the data plane.

Realising the foregoing vision within the constraints of a P4 data plane is, however, far from straightforward: every component of a DNS cache, from storage layout to packet parsing to cache coherence, must be selected from a constrained set of primitives, and each selection carries a specific cost. The next section presents the system and its packet-processing paths; the subsequent design-choice discussion explains the constraints and trade-offs behind them.

3.4 Design and Implementation

This section presents the system architecture, the cache structure (Section 3.4.2), and the parser and packet-handling pipeline (Section 3.4.3). The rationale for the storage, parsing, collision-handling, and update choices follows in Section 3.5.

3.4.1 System Architecture

The system operates as a transparent cache positioned in-line between DNS clients and upstream servers, as illustrated in Figure 3.1. The switch architecture comprises two operational planes: the switch-local control plane[1], which manages the lifecycle of the data-plane process, injects runtime configurations, and collects metrics and logs; and the data plane, which executes forwarding decisions according to the compiled P4 program and manages the cache locally.

[1] The term *switch-local control plane* here refers to the BMv2 daemon and the P4Runtime stub running on the same host as the data-plane process, and is to be distinguished from the remote *Software-Defined Networking* (SDN) control plane discussed in Chapter 2.

The dashed box in Figure 3.1 identifies the ingress pipeline, whose control contains both the `l2_forward` table and the DNS cache logic. Other traffic and DNS packets that fail the cache-processing checks follow the ordinary forwarding path. Eligible UDP port-53 traffic enters the cache logic, which uses a *32-bit Cyclic Redundancy Check* (CRC32)-indexed register array of $N = 10\,007$ slots. The forwarding table selects the upstream output port for cache misses, while cache hits are reflected to the client. Packets then pass through the traffic manager, egress control, checksum update, and deparser. Packets processed by the cache logic recirculate once to remove padding from their reconstructed representation before transmission through the selected egress port (Section 3.4.3).

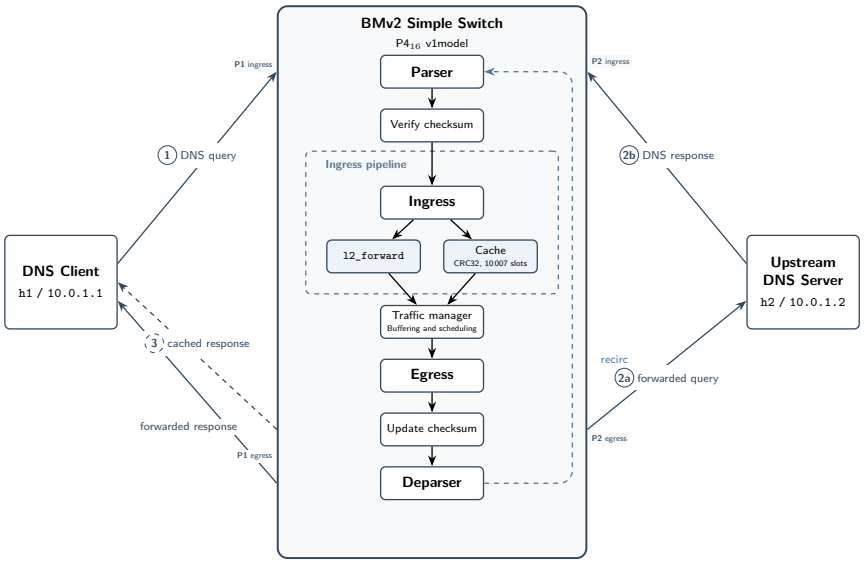


Figure 3.1: DNS caching within the BMv2 v1model architecture.

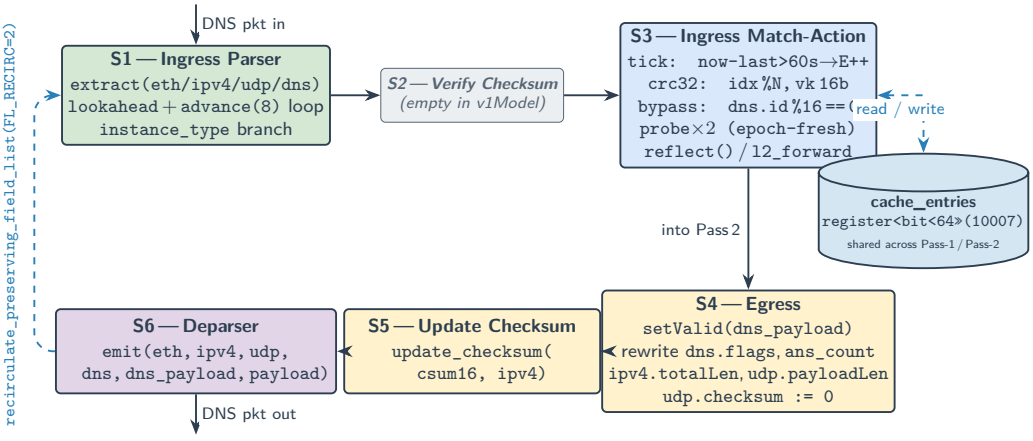


Figure 3.2: Data flow through the P4 ingress pipeline.

The four packet passes labelled in Figure 3.1 together implement the cache protocol. A client query (1) enters at the P1 ingress port and is probed against the cache. On a hit, the ingress synthesises a response in place and the deparser emits it at the P1 egress port back to the client (3)[2], with no upstream traffic generated. On a miss, the query is forwarded out of the P2 egress port to the upstream resolver (2a); the returning response enters at the P2 ingress port (2b), where the ingress writes the resolved record into the cache before the response is delivered to the client at the P1 egress port. Cache-hit replies, forwarded queries, and supported upstream responses each complete one recirculation before leaving the switch.

[2] Before replying to the client, the switch swaps the IP, Ethernet, and port fields, rewrites the DNS header (setting `QR=1`, `RA=1`), and appends an answer record constructed from the cached entry.

The data flow through the P4 pipeline, highlighting the stages where key primitives are applied (parsing, hashing, register access, and packet modification), is illustrated in Figure 3.2.

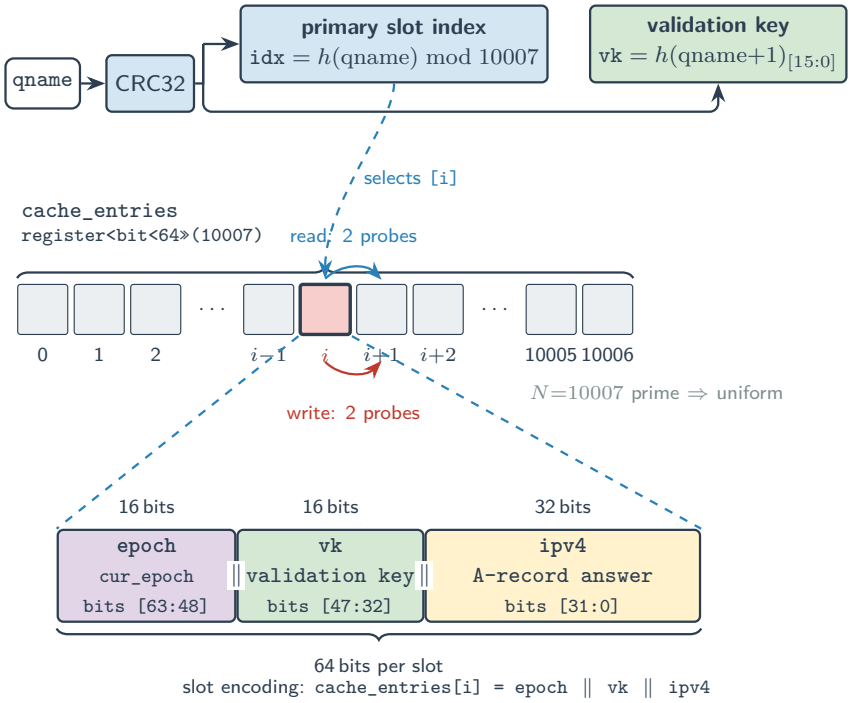


Figure 3.3: Bit layout of a 64-bit DNS cache slot.

3.4.2 Cache Design

The cache is realised as a single register array of fixed-width slots, in keeping with the storage decision of Section 3.5.1. As shown in Figure 3.3, each slot is 64 bits wide and comprises three fields: bits `[63:48]` store a 16-bit epoch counter used for stale-entry invalidation; bits `[47:32]` store a 16-bit validation key (*vk*), computed as the lower 16 bits of a second CRC32 evaluation on the QNAME; and bits `[31:0]` store the resolved IPv4 A-record address. The slot encoding is therefore $cache_entries[i] = epoch \,\|\, vk \,\|\, ipv4$.

Both CRC32 evaluations (*idx* and *vk*) operate on the fixed-width 84-byte metadata buffer containing QNAME, owing to the P4 constraint that bit-string slice indices must be known at compile time. The buffer contains, after right-justification, a sequence of leading-zero padding followed by the actual QNAME.

The array contains $N = 10\,007$ slots, matching the modulus used to derive the primary index. A lookup or write examines at most the primary slot and the next slot. If the primary index is $N - 1$, the next index is 0, as expressed by $(i + 1) \bmod N$.

In Algorithms 1 and 2, n denotes the padded, fixed-width QNAME representation. The placement hash is $H_{idx}(n) = \mathrm{CRC32}(n)$, and the verification hash is $H_{vk}(n) = \mathrm{CRC32}(n + 1)$, where $n + 1$ is fixed-width integer addition to that buffer. Both computations use the same CRC32 algorithm; the

verification hash is not an independent hash family. Its lower 16 bits form the fingerprint vk. The operator $\|$ concatenates fixed-width fields. E is the current 16-bit epoch and Ω the maximum accepted age in epochs: a slot is expired when $age = E - s[63:48] > \Omega$, using 16-bit subtraction.

The prototype sets $\Omega = 0$, so only entries written in the current epoch qualify for a hit. On the first non-recirculated packet arriving more than 60 seconds after the previous tick, the ingress updates the tick timestamp and increments E. This invalidates prior-epoch entries logically; their storage is reclaimed only when a later write probes them. The rule provides a shared aging interval and does not enforce each record's individual DNS TTL. The lookup bypass parameter is $B = 4$.

Algorithm 1: Cache Insert / Update with Bounded Probing and Stale-Cell Reclamation

Input: query name n, IPv4 answer a, current epoch E
Output: $write_success \in \{0, 1\}$

$vk \leftarrow H_{vk}(n)_{[15:0]}$;
$i \leftarrow H_{idx}(n) \bmod N$;
$v \leftarrow E \,\|\, vk \,\|\, a$;
$s \leftarrow \text{cache}[i]$;
$age \leftarrow E - s[63:48]$;
if $s = 0 \;\vee\; s[47:32] = vk \;\vee\; age > \Omega$ **then**
    /* empty, matching fingerprint, or expired */
    $\text{cache}[i] \leftarrow v$;
    return 1;
$i \leftarrow (i + 1) \bmod N$;
$s \leftarrow \text{cache}[i]$; $age \leftarrow E - s[63:48]$;
if $s = 0 \;\vee\; s[47:32] = vk \;\vee\; age > \Omega$ **then**
    $\text{cache}[i] \leftarrow v$;
    return 1;
return 0 ; // collision; $flags \leftarrow 4$

Algorithm 1 handles an upstream A-record response. It first packs the current epoch, fingerprint, and IPv4 answer into a candidate slot value, then tests the primary slot. An empty slot admits a new record; a matching fingerprint permits an in-place update even when the arriving IP is unchanged; an expired slot can be reclaimed regardless of its previous fingerprint. If the primary slot contains a live foreign entry, the same checks are applied at $(i + 1) \bmod N$. A successful write returns 1 immediately. If both slots are unavailable, the algorithm returns 0 and the upstream response still proceeds to the client. Thus admission failure affects future cache hits, while a live foreign entry is never displaced to admit a new record.

Algorithm 2 first applies the transaction-ID bypass rule. With $B = 4$ and uniformly distributed transaction identifiers, one query in sixteen is forwarded upstream even if a cached entry exists, creating an opportunity to observe a changed answer. Otherwise, the algorithm examines the same two slots

Algorithm 2: Cache Lookup with Bounded Probing, Bypass and Epoch Aging

Input: query name n, DNS transaction id tid, current epoch E
Output: (hit, ip)

if $tid \bmod 2^B = 0$ **then**
    /* bypass permits an upstream response to refresh the cache */
    return (miss, $\perp$);
$vk \leftarrow H_{vk}(n)_{[15:0]}$;
$i \leftarrow H_{idx}(n) \bmod N$;
for $k \leftarrow 1$ **to** 2 **do**
    /* probe window of 2 */
    $s \leftarrow \text{cache}[i]$; $age \leftarrow E - s[63:48]$;
    if $s \neq 0 \wedge s[47:32] = vk \wedge age \leq \Omega$ **then return** (hit, $s[31:0]$);
    $i \leftarrow (i+1) \bmod N$;
return (miss, $\perp$);

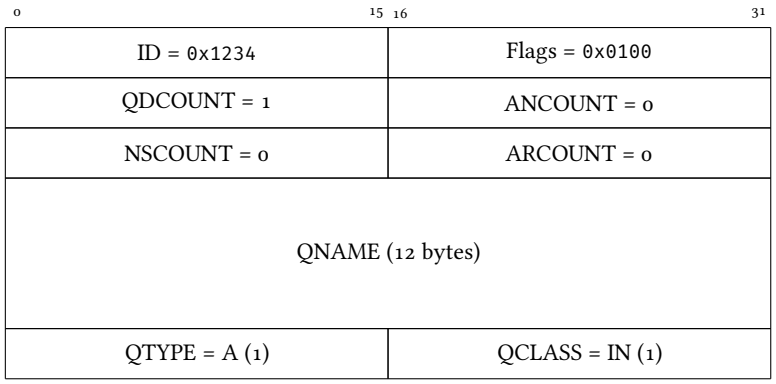
as the write path. A hit requires a non-empty slot, a matching fingerprint, and an age no greater than Ω; the returned address is the slot's lower 32 bits. Failure at the first slot does not terminate the search, since the second may hold the record. If neither qualifies, (miss, $\perp$) denotes a miss with no address and the query follows the upstream path. Fingerprint equality provides a compact key check; distinct names can still produce false matches.

Algorithm 3 isolates the slot-eligibility test used by the write path. An eligible slot can be written; a locked slot holds a live foreign entry. Reclaiming an expired entry is distinct from evicting a live entry, which the cache does not perform.

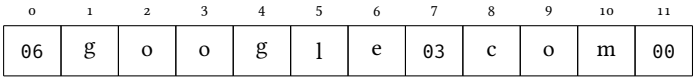
Algorithm 3: DNS Cache Slot Eligibility for Insertion or Update

Input: probed slot value s, validation key vk, current epoch E, age limit Ω
Output: decision $\in$ {Eligible, Locked}

if $s = 0$ **then**
    /* cold slot */
    return Eligible ; // → insert
else if $s[47:32] = vk$ **then**
    /* matching fingerprint */
    return Eligible ; // → in-place refresh
else if $E - s[63:48] > \Omega$ **then**
    return Eligible ; // → reclaim expired slot
else
    /* live foreign entry */
    return Locked ; // never evicted



(a) Wire format of a DNS query for `google.com.` (28 bytes total).



(b) Byte-level encoding of the QNAME field.

Figure 3.4: Wire-format layout of a DNS query for `google.com`.

3.4.3 Implementation

Packet Parsing

A DNS packet comprises a set of fixed-length headers that encapsulate control bits and metadata, along with variable-length fields intended for domain names and corresponding responses. As a hash of the QNAME field serves as the cache key, it must be extracted with precision, ensuring compliance with P4 language constructs and preserving the integrity of message length indicators across protocol layers.

Figure 3.4(a) shows the fixed header and question section; panel (b) details the length-label encoding of QNAME. The variable length of the QNAME field is the structural constraint that necessitates a compile-time-bounded parser strategy, as examined in Section 3.5.2. For instance, consider the domain name `google.com.` illustrated in Figure 3.4: its QNAME representation occupies twelve bytes and consists of two labels followed by the root terminator.

The prototype assumes Type A queries of class IN and advances past QTYPE and QCLASS without checking their values. Algorithm 4 describes first-pass parsing and recirculation; the design alternatives are discussed in Section 3.5.2.

The loops in the listing represent successive transitions through the P4 parser states. The first checks in Algorithm 4 identify IPv4 UDP traffic on port 53 with exactly one question. τ selects between first-pass parsing and recirculation. On the first pass, r tracks remaining bytes, ℓ the accumulated buffer length in bits, *label_len* the characters remaining in the current label, and *qdone* whether the root terminator has been reached. A length octet starts a label; subsequent bytes decrement *label_len*, so a zero byte is treated

Algorithm 4: Data-plane DNS Parser & Answer Re-assembly

Input: packet p, packet length L, instance type τ
Output: metadata (is_valid, is_query, $payload$, $flags$)

$is_valid \leftarrow 0$; $flags \leftarrow 0$;
EXTRACT(eth);
if $eth.type \neq$ IPv4 **then return**;
EXTRACT(ip);
if $ip.proto \neq$ UDP **then return**;
EXTRACT(udp);
if $53 \notin \{udp.src, udp.dst\}$ **then return**;
EXTRACT(dns);
if $dns.qcnt \neq 1$ **then return**;
if $\tau =$ INGRESS_RECIRC **then**
    $r \leftarrow L - 54$;
    for $k \in (64, 32, 16, 8, 4, 2, 1)$ **do**
        if $r \geq k \wedge$ LOOKAHEAD$_{k \cdot 8}(p) = 0$ **then**
            ADVANCE(p, $k \cdot 8$); $r \leftarrow r - k$;
    while $r > 0$ **do**
        EXTRACT($payload$[NEXT]); $r \leftarrow r - 1$;
    return;
$r \leftarrow udp.payloadLen - 20$;
$is_valid \leftarrow 1$; $payload \leftarrow 0$; $\ell \leftarrow 0$; $qdone \leftarrow 0$; $label_len \leftarrow 0$;
$is_query \leftarrow [\, dns.flags = 0x0100 \,]$;
if $dns.flags \notin \{0x0100, 0x8180, 0x8500\}$ **then**
    $is_valid \leftarrow 0$; **return**;
while $r > 0$ **do**
    if $qdone = 0$ **then**
        $d \leftarrow$ LOOKAHEAD$_8(p)$; ADVANCE(p, 8); $r \leftarrow r - 1$;
        $payload \leftarrow payload \,\|\, d$; $\ell \leftarrow \ell + 8$;
        if $label_len > 0$ **then**
            $label_len \leftarrow label_len - 1$;
        else if $d = 0$ **then**
            $qdone \leftarrow 1$; ADVANCE(p, 32); $r \leftarrow r - 4$; $query_len \leftarrow \ell$;
        else
            $label_len \leftarrow d$;
    else
        if $r = 16$ **then**
            ADVANCE(p, 96);
            $ip \leftarrow$ LOOKAHEAD$_{32}(p)$; ADVANCE(p, 32);
            $payload \leftarrow payload \,\|\, ip$; $\ell \leftarrow \ell + 32$; $r \leftarrow 0$;
        else
            $flags \leftarrow 3$; $r \leftarrow 0$;

as the root terminator only at a label boundary. At that terminator, the parser records *query_len* and advances past the four-byte QTYPE/QCLASS fields. The accepted DNS flags distinguish queries from the response forms handled by the prototype. Here, *udp.payloadLen* denotes the UDP length field, including its eight-byte header; subtracting 20 removes that header and the 12-byte DNS header.

For a response with 16 bytes remaining, the answer branch skips the 12-byte resource-record prefix: a two-byte name pointer, two-byte type, two-byte class, four-byte TTL, and two-byte data length. This accounts for the 96-bit advance in the listing. The parser then extracts the four-byte IPv4 address and appends it to the buffer. The constants in this branch encode that restricted layout; the listing is not a general parser for arbitrary resource records or answer chains. An unexpected remaining length sets $flags = 3$, excluding the response from cache interaction.

On recirculation, $L - 54$ excludes the Ethernet, IPv4, UDP, and DNS headers under the assumed header layout. The parser tests zero-padding blocks in descending sizes of 64, 32, 16, 8, 4, 2, and 1 bytes. A block is skipped only when it fits within the remaining length and its lookahead value is zero. The residual payload is then extracted for deparsing. This separates removal of the artificial padding from preservation of the DNS content. For packets admitted to the cache-processing path, the first ingress pass performs the cache lookup or update and selects the forwarding action; the recirculated pass removes padding before the packet is transmitted through the selected egress port.

3.5 Design Choices for Data-Plane DNS Caching

With the system and its processing paths established in Section 3.4, this section explains the choices behind them. Within the capability boundaries of P4 established in Chapter 2, a data-plane-maintained DNS cache must answer four interrelated questions: how cached state is physically represented; how variable-length query names are parsed by a fixed-width pipeline; how hash collisions are handled within a constant per-packet stage budget; and how cache entry state is managed (covering slot displacement under capacity pressure, stale entry invalidation, and in-place refresh) without recourse to the control plane. To render the underlying trade-offs explicit, every subsection concludes with a *Design Decision* box that records the constraint imposed by the substrate, the alternatives considered, the choice adopted in this work, and the cost incurred.

3.5.1 Storage Primitive

The first decision concerns the physical container for cached records. The P4 V1model architecture exposes three candidates: match-action tables, register arrays, and `extern` objects (P4 Language Consortium, 2024)[3].

[3] The discussion in this subsection is restricted to the v1model architecture.

Match-action tables are populated and updated by the control plane and therefore cannot be written from the data plane during packet processing; they additionally support wildcard semantics that DNS lookups do not require [4].

[4] *Ternary Content-Addressable Memory* (TCAM) was designed for wildcard or masked matching, using "don't care" bits to specify ternary conditions.

Register arrays, on the contrary, are addressed by index and may be both read and written within the same pipeline traversal [5], at the cost of constant-width slots, an absence of associative matching, and potentially additional latency arising from the sequential dependence between read and write within the same stage.

[5] The `register` behaviour differs across architectures.

`extern` objects encapsulate vendor-specific functionality and trade portability (the level of difficulty in porting to different targets) for capability. In the v1model architecture package, the term `extern` here refers specifically to vendor-defined externs beyond the standard `register`, `counter`, and `meter` family.

For a cache that must be updated entirely on the data plane and that requires only exact-match lookups, the register array is the only candidate that satisfies both constraints.

Design Decision 3.1: Storage primitive

Constraint: Cache writes must occur on the per-packet path without control-plane involvement.

Alternatives: match-action table (control-plane writes only, supports wildcards); register array (data-plane read/write, fixed-width slots); `extern` object (vendor-specific, non-portable).

Choice: Register array, indexed by a hashed slot identifier, with constant-width key and value fields.

Cost: Total capacity is bounded by available *Static Random-Access Memory* (SRAM) resources within the *Match-Action Unit* (MAU) pipeline, and wildcard or longest-prefix matching is unavailable within the cache structure itself.

3.5.2 Parser Strategy

The QNAME field of a DNS query is a sequence of length-prefixed labels of variable total length up to 255 bytes, with each label not exceeding 63 bytes.

The parameters (e.g. index) of bit string operations in P4, such as bit slicing `value[hi:lo]` and data access `pkt.lookahead<T>()` must be compile-time known constants (P4 Language Consortium, 2024), and the parser must therefore be designed to accommodate the variable-length QNAME within a fixed number of states and transitions.

Three approaches to handling variable-length QNAMEs illustrate the parser trade-off:

1. A byte-by-byte finite-state automaton, in which a single state advances the parser one byte at a time, minimises lines of code but generates an excessive number of state transitions per packet.
2. Kaplan and Feibish (2022) use states with fixed extraction widths of 2^k bytes. After reading a label's length, the parser selects widths whose sum equals that length, reducing transitions without requiring a runtime-variable extraction width.
3. J. Kim *et al.* (2021) parses the QNAME field using four fixed-width states of 15 bytes each, achieving structural simplicity at the price of truncating any name longer than 60 bytes. This was considered due to the constraint of the *Tofino Native Architecture* (TNA) architecture.

For example, the DNS name `www.google.com` is encoded as `[3]www[6]google[3]com[0]`, where each bracketed value denotes a one-byte length field and the final zero terminates the name.

A power-of-two parser first reads the length of `www`: $3 = 2 + 1$, so it extracts `ww` and then `w`. For `google`, $6 = 4 + 2$, so it extracts `goog` and then `le`; for `com`, $3 = 2 + 1$, so it extracts `co` and then `m`. After each extraction, the parser subtracts the consumed width from the remaining label length and selects the next fitting power of two. This requires six character-extraction steps instead of twelve. The method reduces state visits at the cost of more states and transitions in the program, with coverage still bounded by the configured label count and maximum length.

The prototype uses two distinct parsing procedures. QNAME extraction remains byte-by-byte, with a counter tracking the remaining characters in each label. On recirculation, padding removal uses descending power-of-two lookahead and advance operations to skip leading zero blocks, as shown in Algorithm 4. Thus the binary-decomposition optimisation applies to padding removal, while QNAME extraction retains its per-byte state visits. The resource evaluation compares this implementation with a byte-loop padding-removal pilot on BMv2.

Design Decision 3.2: QNAME extraction and padding removal

Constraint: The parser must commit to a finite-state structure at compile time and is bounded by the per-packet stage and *Packet Header Vector* (PHV) budget.

Alternatives (QNAME): four 15-byte fixed windows (J. Kim *et al.*, 2021); byte-by-byte automaton; 2^k-stride label extraction (Kaplan & Feibish, 2022).

Alternatives (padding): byte-by-byte zero stripping; fixed-width binary decomposition.

Choice: Byte-by-byte QNAME extraction with length-label bookkeeping; fixed-width binary decomposition for zero-padding removal on recirculation.

Cost: QNAME extraction incurs per-byte state visits; padding removal requires several fixed-width states. Supported names remain bounded by the configured buffer and parsing limits.

3.5.3 Collision Handling

Cache slots are addressed by hashed indices, and collisions are unavoidable once the load factor becomes non-trivial. The strategies considered are linear probing, bounded linear probing as developed in TEA (Table Extension Architecture) (D. Kim *et al.*, 2020), and bounded linear probing augmented with a compact per-slot verification fingerprint.

The cache uses two-slot bounded linear probing: the primary slot is $H_{idx}(n)$ mod N, and the only alternative is its successor modulo N. CRC32 of the incremented QNAME buffer supplies a 16-bit verification fingerprint stored in bits `[47:32]` of each slot. This avoids storing the full QNAME in every entry, while allowing the read and write paths to distinguish most unrelated records within the probe window. If both candidate slots contain live foreign entries, the new record is left uncached. A d-left scheme instead partitions storage into d subtables and gives each key a candidate in each partition; this provides more placement choices but requires more hash and memory operations.

Design Decision 3.3: Collision handling

Constraint: Collision resolution must complete within a fixed number of pipeline stages, avoiding unbounded probing.

Alternatives: linear probing; bounded linear probing (D. Kim *et al.*, 2020); bounded probing with hash fingerprint; d-left hashing.

Choice: Two-slot bounded linear probing with a 16-bit *Cyclic Redundancy Check* (CRC) verification fingerprint.

Cost: Admission can fail within the bounded probe window as occupancy increases. Verification requires a second hash computation, and a 16-bit fingerprint can falsely match a different name.

3.5.4 Cache Update Model

The fourth decision concerns how cache entry state is managed across three distinct forms of change, each arising under different conditions.

Eviction arises when all candidate probe slots are occupied by entries for *different* domains and a newly arrived entry would need to displace one of them. *Least Recently Used* (LRU) and *Least Frequently Used* (LFU) require per-entry metadata updated on every access, which is prohibitively expensive within the stage and SRAM budget of a P4 pipeline; Zhao *et al.* (2023) show that an LRU approximation is realisable on the data plane, but at non-trivial stage cost. One greedy alternative is to evict the entry at the primary slot whenever the second probe position is occupied by a different domain during response processing. This greedily favours the most recent entry but risks displacing a live record for a popular domain that happens to share the same probe set. The present implementation deliberately forgoes an eviction policy: when all candidate slots contain live foreign entries, the arriving entry is left uncached and the query falls through to the upstream server, mitigating the overhead of accessing and maintaining the per-entry access metadata.

Invalidation occurs when an existing entry is found to have exceeded its TTL. Maintaining a per-entry timestamp and comparing it against the current time on each lookup is expensive in terms of both state and per-packet processing. An active control-plane sweep can enforce TTL expiry globally, but reintroduces control-plane involvement and couples liveness to the sweep interval. The present implementation adopts an epoch-based strategy: a shared epoch counter advances the aging reference for all entries, while each probed slot is judged against the age limit Ω in Algorithm 2. Invalidation is physically lazy: expired slots are not cleared eagerly but can be overwritten by a subsequent write.

Refresh occurs when an upstream response arrives for a domain *already present* in the cache. A matching fingerprint permits the response to overwrite the stored address and epoch, whether or not the IP address has changed. The control-plane-driven model exemplified by P4DNS (Woodruff *et al.*, 2019), in which the control plane routes cache updates into the match-action tables, falls outside the data-plane-only constraint adopted here.

One alternative is to probabilistically bypass the cache on outgoing queries: the data plane checks whether the transaction identifier satisfies TXID mod $M = 0$ for a predetermined modulus M and, when the condition is satisfied, bypasses the cached record and forwards the query upstream. The resulting upstream response can then overwrite

a matching entry and update its epoch. This passive strategy requires no per-packet control-plane involvement and is viable under the assumption that a modest staleness window is tolerable, as argued in Section 3.3.

Design Decision 3.4: Cache update model

Constraint: Invalidation and refresh must execute entirely within the data plane without per-packet control-plane involvement.

Alternatives (eviction): LRU (Zhao *et al.*, 2023); LFU; greedy primary-slot displacement on second-probe conflict; no eviction (upstream fallthrough on full probe set).

Alternatives (invalidation): active control-plane sweep; per-entry lazy TTL comparison; epoch-based global invalidation; pre-populate heavy hitters with no invalidation.

Alternatives (refresh): control-plane table installs (Woodruff *et al.*, 2019); TXID-modulo probabilistic cache bypass; response-triggered in-place overwrite; combined TTL/probabilistic schemes.

Choice: No eviction policy is implemented; full-probe-set conflicts fall through to the upstream server. Epoch-based stale entry invalidation with $\Omega = 0$. TXID-modulo bypass and response-triggered in-place overwrite for matching fingerprints.

Cost: Without eviction, full probe sets result in uncached entries until a conflict resolves; records may persist beyond their nominal TTL until the epoch boundary; refresh depends on an upstream response reaching an eligible slot.

The following evaluation measures their resource and processing costs on BMv2 and compares the register-based cache with a match-action baseline.

3.6 Evaluation

The evaluation examines the resource and processing costs of the system and the trade-offs identified in the preceding design discussion:

- The static resource analysis characterises the register storage, parser states, and table entries required by the implementation on BMv2 (Section 3.6.2).
- The cache analysis examines admission failures and deliberate lookup bypass, while the per-packet statistics quantify pipeline traversals and parser-state visits (Sections 3.6.3 and 3.6.4).
- The cache-hit latency comparison characterises the observed timing cost of the two implementations on the same software target (Section 3.6.5).

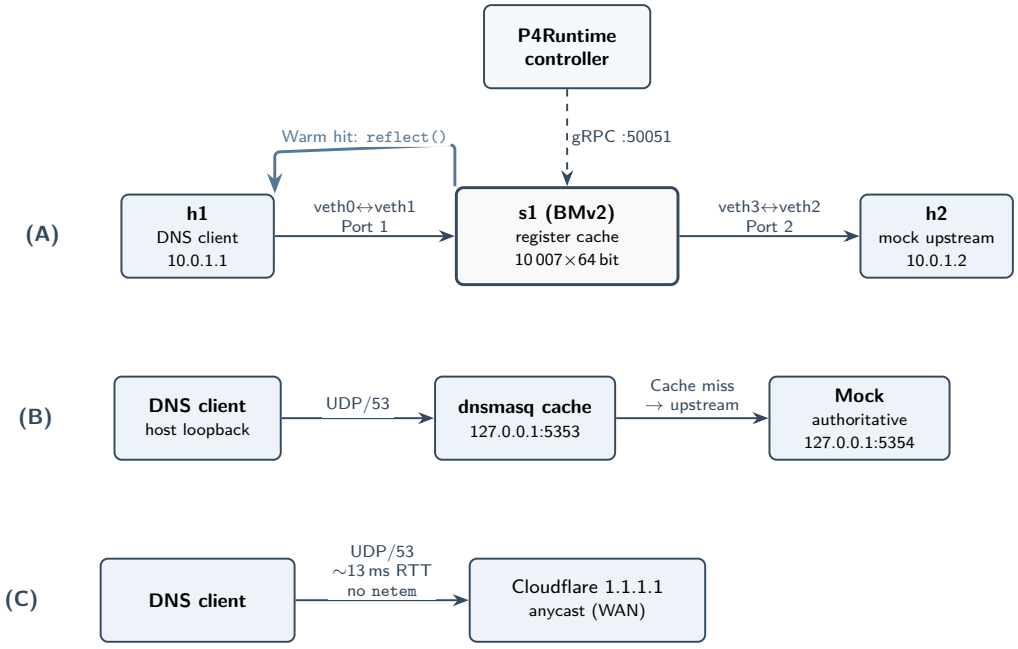


Figure 3.5: The single-host topology used for evaluation.

3.6.1 Experimental Setup

The prototype evaluated in this chapter is implemented for the BMv2 `simple_switch` target compiled from the `v1model` architecture. The benchmark uses a single-host topology with client-side and upstream-side endpoints connected to the BMv2 instance through virtual Ethernet pairs, following the topology in Figure 3.5. The host machine has an Intel Xeon Bronze 3204 *Central Processing Unit* (CPU) and 40 GB of *Random Access Memory* (RAM). The query set comprises the top 1 000 domains extracted from Cisco Umbrella's top 1-million dataset ('Cisco Popularity List', 2016), reflecting the cache's purpose of offloading heavy-hitter DNS lookups.

To compare with the *Match-Action Table* (MAT)-based design, the P4DNS implementation of Woodruff *et al.* (2019) is re-implemented on BMv2 with a control-plane-driven update model, and the same topology and workload are used for evaluation. All query transaction identifiers (TXIDs) are drawn uniformly at random from the full 16-bit space, ensuring that the TXID-modulo probabilistic bypass described in Section 3.5.4 fires at the expected rate.

The latency benchmark sends frames through persistent raw sockets and captures kernel packet timestamps. Queries and responses are matched by transaction identifier, QNAME, the query/response flag, UDP ports, capture interface, and packet direction. For each cache hit, latency is calculated as $t_{\text{response}} - t_{\text{query}}$ between the outgoing query and the returning response at the client-side interface. Forwarded queries are recorded separately from local cache hits. This interval includes the software-switch and virtual-

Table 3.1: Static P4 resource usage for the register-based and MAT-based DNS cache implementations on BMv2.

Metric	Register-based	MAT-based	Unit
Pipeline complexity (lower is simpler)			
`parser_states`	67	41	states
`actions_total`	27	7	actions
`tables_total`	25	5	tables
`conditionals_total`	16	4	conditionals
Storage cost (cache + epoch state)			
`register_arrays`	3	0	arrays
`register_cells_total`	10 009	0	cells
`register_bits_total`	640 512	0	bits (≈78 KiB)
`mat_cache_entries`†	0	10 007	entries

interface processing, while excluding sender-socket initialisation and Python callback scheduling from the timestamp boundaries.

Each design is measured in complete run with ten warm queries per domain, issued sequentially. The MAT cache is populated with all 1 000 entries before querying; the register-based cache receives a synthetic upstream response for each domain immediately before its ten queries. Both use the A-record value `1.2.3.4`. This latency workload is separate from the static admission experiment in Section 3.6.3.

Static resource counts are extracted from the compiled P4 artifacts, while instrumented runs record pipeline traversals and parser-state visits. The sample sizes for the processing-statistics comparison are reported in Table 3.3.

3.6.2 Resource Usage

As shown in Table 3.1, the ingress pipeline of the register-based design is computationally more complex than that of the MAT-based design due to the state handling of cache access and update, whilst the MAT-based design can simply apply the match-action unit to the DNS query but relies on control-plane insertion and cannot perform data-plane writes without control-plane intervention.

In BMv2, control flows are implemented by a compiler-generated table graph, which inflates the count of synthetic tables in the register-based design. The `tables_total` and `conditionals_total` counts therefore include compiler-generated auxiliary tables (e.g., the `tbl_act` series), not only those declared in the source via the `table` keyword, and reflect the per-stage complexity of the compiled pipeline. Baseline forwarding resources (e.g., L2 forwarding tables) are excluded from the comparison.

Table 3.2: Probe-depth distribution for insertions and warm lookups under the two-probe linear scheme, including the probabilistic bypass fraction.

Event	Count	Share	Comment
Insert: attempt-1 (slot `idx`)	949	94.90 %	Lands directly at the primary slot.
Insert: attempt-2 (slot `idx+1`)	42	4.20 %	Recovered by one linear probe.
Insert: admission-fail	9	0.90 %	Both `idx` and `idx+1` already occupied.
Lookup: hit-1	~890	88.97 %	94.9 % slot-1 hits × 15/16 non-bypass.
Lookup: hit-2	~39	3.94 %	4.2 % slot-2 hits × 15/16 non-bypass.
Lookup: miss (collision)	~8	0.84 %	0.9 % admission-fail × 15/16; never cached.
Lookup: miss (bypass)	~62	6.25 %	Forced upstream by `dns.id[3:0]==0`[3].
Final cache occupancy	991/10 007 = 9.90 %		= 949 + 42.

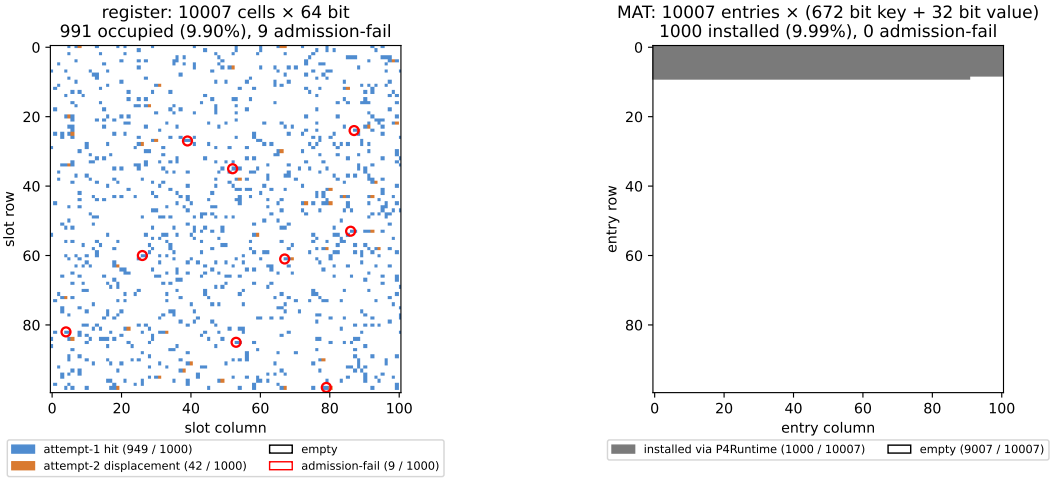


Figure 3.6: Cache occupancy of the register-based and MAT-based designs.

3.6.3 Cache Hit Ratio

Table 3.2 gives the analytical per-event breakdown of the same warm-populate run whose spatial outcome is shown in Figure 3.6.

Figure 3.6 shows the spatial distribution of valid entries across both cache implementations after the warm-populate phase, in which the top 1 000 domains of the Cisco Umbrella ('Cisco Popularity List', 2016) trace are inserted prior to the query sequence.

In the register-based cache, occupied slots are scattered across the 10 007-cell array with no detectable clustering, confirming that the CRC32 algorithm produces a near-uniform placement distribution. Of the 1 000 insertion attempts, 949 were placed at their primary slot and 42 required a single displacement step to the adjacent probe position, yielding 991 successfully occupied slots at a load factor of 9.90 %. The remaining 9 entries failed admission entirely: both their primary and displaced slots were already occupied within the probe window, as indicated by the red-circled markers.

The MAT-based cache presents no admission failures, since exact-match table insertion is unconditional within available capacity. The register-based

Table 3.3: Pipeline traversals and parser-state visits per packet.

Mode	passes / pkt	state entries / pkt	total state entries
Register-based (byte loop, pilot N=20)	2.00	331.30	6 626
Register-based (binary decomp, N=2 000)	2.00	224.31	448 627
MAT-based (N=2 000)	1.00	157.97	315 944

design achieves data-plane learning without control-plane intervention at the cost of a small, bounded admission-failure rate, whereas the MAT-based design delegates all insertion to the control plane.

The cache hit ratio in this scenario is a micro-benchmark result, which must not be read as a prediction of the cache's deployment hit ratio. Each preloaded domain is queried ten times in immediate succession. There is no churn, no TTL-driven invalidation or rank-skewed access pattern. Under these conditions, excluding queries deliberately bypassed by the TXID-modulo rule, steady-state cache misses arise from admission failures under the chosen CRC32 hash function and two-slot probe window. The bypass rule adds a separate expected 6.25 % of queries forwarded upstream, as shown in Table 3.2.

3.6.4 Processing Cost Analysis

The byte-loop pilot used for comparison implements a byte-level loop that consumes one byte per state, which is simple but incurs a high state count overhead. As shown in Table 3.3, following the approach of Kaplan and Feibish (2022), the enhanced binary-decomposition parser achieves the same function with a logarithmic number of states by consuming up to 64 bytes per state and branching on the residual length. The reported averages differ by approximately one-third, but the byte-loop pilot contains only 20 packets whereas the binary-decomposition run contains 2 000; the unequal sample sizes make this an indicative comparison.

The register-based cache makes two pipeline traversals for packets admitted to its cache-processing path, compared with one for the MAT-based baseline. The first traversal performs lookup or update and selects the forwarding action; recirculation removes padding before transmission. Epoch maintenance additionally accesses the `last_tick_ts` and `cur_epoch` registers. These operations account for part of the processing cost of keeping cache maintenance within the data plane.

The statistics distinguish the work performed by the two implementations. Its effect on elapsed latency depends on the execution target, scheduling, and memory-access behaviour; parser-state visits and pipeline passes therefore provide processing-cost evidence rather than a direct timing comparison.

Table 3.4: Local cache-hit latency on BMv2 (ms).

Design	Hit samples	p50	p95	p99
Register-based	9 371	14.776	18.872	21.900
MAT-based	10 000	10.070	15.153	16.907

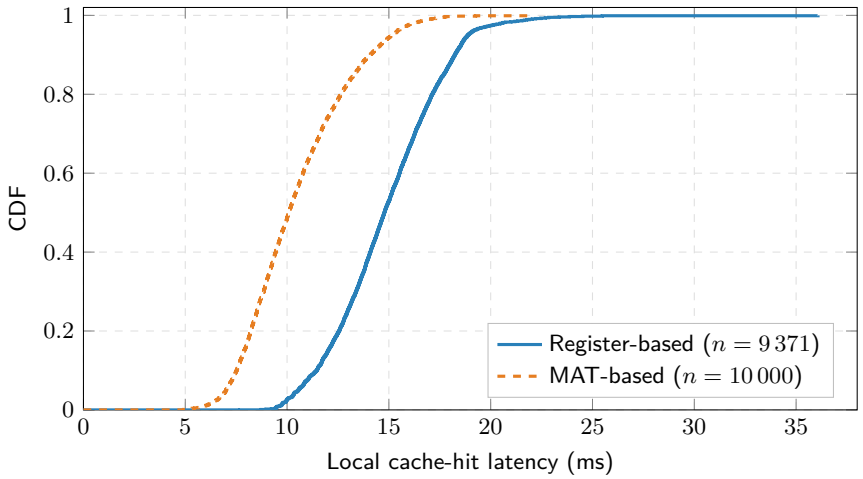


Figure 3.7: Local cache-hit latency CDF for the two cache designs.

3.6.5 Latency Analysis

The latency workload produces 9 371 local cache hits for the register-based design and 10 000 for the MAT-based design. The register-based run additionally records 619 TXID-modulo bypasses and ten other forwarded queries. The latter occur in four groups approximately 60 seconds apart, consistent with the shared epoch advancing during a domain's query sequence. They are reported separately from static admission failures. Neither run records a timeout.

Table 3.4 and fig. 3.7 summarise only local cache-hit responses; forwarded queries are excluded from both distributions.[6] In this run, the register-based median is 4.706 ms higher than the MAT-based median, an increase of approximately 46.7 %. Its p95 and p99 latencies are also higher. The separation is consistent with the additional processing described in Section 3.6.4, but the implementations differ in parsing, lookup, and state maintenance as well as recirculation, so the comparison does not isolate the cost of any one operation. The results describe this sequential workload on the measured host.

[6] An earlier CDF used a timer started before each send call, including raw-socket initialisation. A no-transmission diagnostic on the same host reproduced approximately 16, 20, and 24 ms modes during socket binding, explaining the prominent steps. The present curves use outgoing and returning packet timestamps.

3.7 Discussion and Summary

3.7.1 Lessons for P4 Function Development

Self-contained data-plane services require state representation, packet processing, and maintenance to be designed together. Bounded parsing and fixed-width storage determine the supported inputs, while admission and refresh policies determine service behaviour under capacity and freshness constraints. The DNS cache illustrates these dependencies through the costs identified in Section 3.5.

The recirculation cost applies to packets admitted to the cache-processing path. Each undergoes two pipeline traversals: the first performs cache lookup or update and selects the forwarding action, while the second removes padding before transmission. Other traffic and DNS packets that fail the cache-processing checks are forwarded directly without this additional traversal.

The probabilistic update model, in turn, produces a non-zero probability that a cached record persists beyond its nominal TTL. As there is no mechanism for checking the authenticity of an incoming response, the cache may be poisoned by an adversary who injects a response with a malicious host IP address.

Fixed-width state representations constrain service coverage. In the DNS cache, constant-width register slots enable packet-triggered writes, but accommodate only record formats whose key and value fit within the chosen widths. Widening all slots to accommodate larger records multiplies SRAM consumption proportionally; delegating variable-width entries to the control plane instead violates the data-plane-ownership requirement.

The collision-handling and update policies introduce a residual effectiveness floor that probabilistic refresh tuning cannot compensate. Within the bounded two-slot probe window, insertions fail silently when both the primary and the adjacent slot contain live foreign entries. No live entry is evicted, so the cache can learn new entries when an empty or expired slot is probed, or refresh an existing entry in place; otherwise, the new record is rejected and the old record persists until it expires at the epoch boundary.

These costs share a common structure: each is a direct consequence of restricting the per-packet path to operations that complete within a bounded number of pipeline stages, using only fixed-width state and without recourse to per-entry metadata or control-plane coordination.

The need for bounded representations and repeated parser-state visits also affects stateless parsing and packet-rewriting functions. On hardware, these functions must fit target-specific header storage, pipeline-stage, memory-access, and recirculation budgets (Section 2.4.1).

These costs make complex or nested *Type-Length-Value* (TLV) decoding a difficult fit for resource-constrained P4 hardware pipelines. Host-based *Extended Berkeley Packet Filter* (eBPF) permits more flexible packet traversal, combining verifier-safe kernel checks with deeper user-space decoding (Chapter 4).

3.7.2 Limitations

Three limitations qualify the findings reported in this chapter: the scope of supported protocols, the generalisability of the BMv2 measurements to hardware targets, and the representativeness of the evaluation workload.

Protocol scope The cache handles IPv4-based A records whose QNAME wire encoding does not exceed 64 bytes. Record types that require variable-length or multi-record responses (such as AAAA, CNAME, and MX) are not supported, nor are *DNS Security Extensions* (DNSSEC)-signed responses. Extending coverage to these types would require either relaxing the fixed-width constraint imposed by the P4 header model or delegating record assembly to the control plane, either of which increases design complexity.

Platform generalisability The evaluation uses the BMv2 software target. Mapping the design to hardware requires checking target-specific parser, pipeline-stage, memory-access, and recirculation budgets (Section 2.4.1); the software-target latencies do not establish hardware timing. The admission-failure rate of the register-based cache depends on the CRC32 hash function and the two-slot probe window; these parameters may require re-tuning on targets with different SRAM layouts or access widths.

Workload representativeness The experiments select the top 1 000 Cisco Umbrella domains and query each ten times, rather than reproducing their relative popularity in live traffic. The static admission experiment does not model TTL-driven invalidation or cache churn, while the latency run populates the register-based cache immediately before each domain's queries. These workloads characterise specific cache paths rather than deployment hit ratios under changing working sets.

3.7.3 From Data-Plane to Shared State Ownership

The DNS cache shows how pipeline-compatible state representations and update operations enable self-contained service processing.

When state maintenance or analysis exceeds the chosen implementation's capabilities, these tasks can be divided between the data plane and end-host

software. Recent work demonstrates that programmable switches can support demanding state maintenance (Scazzariello *et al.*, 2023); TEA, for example, extends switch memory through access to server *Dynamic Random-Access Memory* (DRAM) via *Remote Direct Memory Access* (RDMA) (D. Kim *et al.*, 2020). This approach requires hardware and deployment support that differs from commodity end-host processing.

Chapter 4 examines a different division of responsibility through *Generic Object Oriented Substation Event* (GOOSE) anomaly detection: the kernel maintains sequence history and performs bounded per-packet checks, while user space aggregates events and updates enforcement policy. This extends the investigation from the costs of pipeline-resident maintenance to those of software coordination, addressing RQ2.

3.7.4 Summary

This chapter has presented a fully data-plane-resident DNS cache, realising its storage and update mechanisms within the resource budget of an experimental P4 target. The systematic analysis of the design choices conducted in Section 3.5 renders the underlying trade-offs explicit and addresses research question RQ1 by characterising the costs of control-plane-independent DNS caching in terms of pipeline complexity, storage requirements, per-packet processing, cache-hit latency, and restrictions on cache admission, invalidation, refresh, and supported record formats. The division of maintenance responsibilities introduced in Section 3.7.3 provides the point of departure for Chapter 4, which examines the costs of coordinating kernel checks with user-space policy maintenance.

Chapter 4

GOOSE Detection with Shared State Ownership

4.1 Overview

This chapter examines how processing and runtime state maintenance can be partitioned between the data plane and user space while retaining an independent packet-processing fast path. Following the data-plane-owned *Domain Name System* (DNS) cache in Chapter 3, it addresses RQ2 through a stateful *Generic Object Oriented Substation Event* (GOOSE) analysis function implemented with *Extended Berkeley Packet Filter* (eBPF)/*eXpress Data Path* (XDP) on a commodity host. The prototype provides evidence for the second thesis claim (*S1*, Section 1.4): user-space coordination can extend data-plane processing through shared state maintenance while preserving data-plane independence. The measurements characterise per-packet processing overhead and the subsequent reaction delay before enforcement.

The main deliverable is a data-plane function for GOOSE analysis: bounded protocol checks and sequence-state updates execute in the kernel, while user space aggregates anomaly events and maintains enforcement policy. Integration with a transport–energy digital-twin co-simulation is presented as a proposed use case for this function. Its state-exchange interface supports future physical-context assessment.

The remainder of this chapter is organised as follows. Sections 4.2–4.4 motivate GOOSE analysis, explain the communication scenario, and define the threat model. Sections 4.5 and 4.6 present the function design, its co-simulation integration use case, and the prototype implementation. Sections 4.7 and 4.8 examine attack scenarios and report the architectural comparison and indicative measurements. Section 4.9 discusses the implications for stateful function design and summarises the chapter.

4.2 Motivation

Stateful traffic analysis uses cross-packet history or flow-level context to identify deviations from normal behaviour (Chen *et al.*, 2023; Scazzariello *et al.*, 2023; Teng *et al.*, 2022). Its design must determine which state-maintenance responsibilities the data plane can sustain and which require software coordination. Three properties of GOOSE make it a demanding case for examining this division, and together motivate the in-kernel approach developed in the following sections.

First, IEC 61850 mandates that protection-relay subscribers act on a GOOSE trip notification within a tight millisecond-scale latency budget; retransmission bursts are designed explicitly to deliver each state change within this window (Hoyos *et al.*, 2012). Any inline detection mechanism must therefore complete per-packet processing within a fraction of this budget. A conventional user-space approach introduces extra latency overhead that is incompatible with this requirement: earlier work by the author and colleagues (Holik, Cook *et al.*, 2025) compared user-space, *Software-Defined Networking* (SDN), and in-kernel eBPF handling of GOOSE on an emulated substation testbed (Figure 4.1), where the user-space and SDN paths exceeded the millisecond budget whilst the in-kernel eBPF path remained within it.

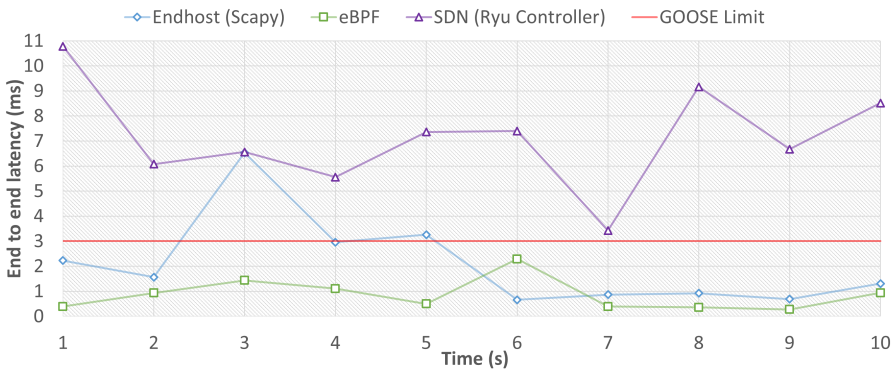


Figure 4.1: End-to-end GOOSE latency across handling paths on an emulated testbed (Holik, Cook *et al.*, 2025).

Second, the GOOSE frames considered in this chapter carry cleartext data without authentication. Its two principal header counters follow predictable, observable semantics: `stNum` is a per-stream state counter incremented on each data-set state change, whilst `sqNum` resets to zero on every such change and increments only within the retransmission burst that follows it. Both progressions are therefore fully predictable to any passive observer on the substation *Local Area Network* (LAN). An adversary with network access to the LAN can therefore craft syntactically valid spoofed or replayed frames without advanced knowledge, making GOOSE a natural and well-documented attack surface (Girdhar *et al.*, 2025; Hoyos *et al.*, 2012).

Third, protocol-state analysis and physical-process assessment require dif-

ferent kinds of information. Fast per-packet checks (counter monotonicity, inter-arrival timing, and source legitimacy) can be performed by an in-kernel XDP program, before frames arrive at the kernel network stack, at a per-packet cost that stays within the protocol's timing budget (Lapolli *et al.*, 2019; Vieira *et al.*, 2020). However, attacks that are tailored to the physical process (for example, a valid `stNum` increment that contradicts the transport-system co-simulation's current state) demand cross-system context available only in user space. This motivates a bounded in-kernel analysis function with asynchronous user-space policy maintenance, and a separate integration interface through which external context could later inform policy.

4.3 GOOSE Communication and Detection Background

4.3.1 Protocol Role in the Explored Scenario

In a digital substation, publisher and subscriber roles are performed by *Intelligent Electronic Devices* (IEDs) connected to the same layer 2 network, including protection relays and devices associated with breaker control. A publisher multicasts a configured data set containing process measurements or status indications; subscribing devices use the relevant values to coordinate protection and control, such as communicating a fault indication or a trip command (Girdhar *et al.*, 2025; Hoyos *et al.*, 2012). The functional goal is timely distribution of a process-state change to the devices that must respond to it. Repeated transmission of the current data set supports delivery and allows subscribers to monitor whether the publisher remains active, which can be seen as a heartbeat mechanism.

The integration scenario in this chapter represents these roles with software publishers and a software subscriber. Two publishers represent separate grid elements, while the subscriber maintains their reported supply state for the transport-side charging simulation. Each publisher's application data set contains rated power, reported output power, and a *Coordinated Universal Time* (UTC) timestamp. The subscriber associates the data with the corresponding grid element and treats zero output with a positive rated power as a tripped supply. It exports the resulting power and status to the transport simulation over *Transmission Control Protocol* (TCP). A simplified software model generates these values.

GOOSE therefore carries the energy-side supply information between the emulated IEDs; TCP carries the aggregate state exchanged with the transport simulator. The XDP function observes the GOOSE communication and checks its sequence and timing behaviour. The trip-command and overcurrent examples motivate the threat analysis through their potential substation consequences.

4.3.2 Protocol State and Prior Detection Approaches

IEC 61850 GOOSE is a Layer-2 multicast protocol designed for time-critical messaging in digital substation automation. Each frame carries an Application ID (APPID) field used to distinguish configured application streams, a state number (`stNum`) that increments on each data change, and a sequence number (`sqNum`) indexing the retransmission burst that follows every state transition. The `sqNum` also acts as a heartbeat mechanism, whilst `stNum` serves as a version number for the data set (Hoyos *et al.*, 2012).

The unauthenticated, cleartext GOOSE communication considered here reflects the assumption of a physically isolated LAN (Williams, 1994) but creates a well-documented attack surface: because `stNum` and `sqNum` follow predictable semantics, an adversary with passive medium access can craft convincing spoofed or replayed frames (Girdhar *et al.*, 2025). The attack surface arising from these predictable counter semantics is noted here; the threat model is defined in full in Section 4.4.

Existing approaches to GOOSE anomaly detection operate predominantly in user space. Rule-based engines and offline machine-learning pipelines coupled to SDN controllers both require packet copies to traverse the kernel network stack before reaching the detection logic (Yegorov *et al.*, 2023); the resulting latency is incompatible with the strict response budget that substation protection demands, as discussed in Section 4.2. Recent work has pushed detection into an eBPF software data plane through adaptive in-network feature quantisation (Mahboob *et al.*, 2025), reducing forwarding-path overhead. External co-simulation state provides a possible additional input to user-space policy, motivating the use case presented here. Holik, Mcilwraith *et al.* (2025) presents an encryption *Virtual Network Function* (VNF) for protecting GOOSE messages.

A potential source of this cross-system context is a *Digital Twin* (DT), a software model that continuously mirrors the state of a physical or cyber-physical system (Minerva *et al.*, 2020). In the transport-energy nexus, a co-simulation *Digital Twin* (DT) can expose mobility and *Electric Vehicle* (EV)-charging state that substation-local packet inspection cannot observe. The integration interface in Section 4.5.2 provides access to this state.

4.4 Threat Model

The threat model assumed throughout this chapter reflects the Purdue network-segmentation assumption (Williams, 1994): the substation LAN is nominally isolated, so an adversary who penetrates this isolation is assumed to have physical access to the GOOSE communication link (layer-2 access on the substation LAN), achievable through compromise of an on-site host or through an insider threat. The attacker is capable of injecting, modifying, or

replaying Ethernet frames at the GOOSE EtherType (`0x88B8`). The attacker has domain-specific knowledge, is familiar with `stNum`/`sqNum` semantics and able to construct syntactically valid frames based on sniffing. The attacker is not assumed to know the detection system's per-source baselines or policy thresholds. The objectives of a successful attack include triggering an unexpected breaker trip, suppressing a legitimate message from being processed, or denying service by exhausting the sequence-counter state of a legitimate IED; each of these propagates consequences into the broader transport-energy nexus through the EV charging infrastructure it supervises (Girdhar *et al.*, 2025).

4.5 Function Design

The main system comprises an in-kernel GOOSE analysis function and asynchronous user-space policy support. Its input is the Ethernet frame stream at the XDP hook; its outputs are a per-packet pass/drop action, updated protocol state, and anomaly records for user space. The function runs without a synchronous request to the policy component or the simulator. The co-simulation integration is described separately in Section 4.5.2.

4.5.1 GOOSE Analysis and Policy Maintenance

Module decomposition. The user-space component of the architecture is structured as a single long-running supervisor process hosting the following components:

- *Event Consumer* that drains the kernel ring buffer, reading and processing anomaly events emitted by the in-kernel detector.
- *Policy Updater* that mutates eBPF maps, populates the deny list, and generally translates detection outcomes into enforceable policy.

Composition. The design combines bounded protocol parsing and per-flow state updates in XDP with event aggregation and policy maintenance in user space. The *Berkeley Packet Filter* (BPF) maps and ring buffers provide the coordination boundary: packets consult the currently installed policy, while anomaly events can cause later updates. The simulation client adds application-specific context through the same interface.

The prototype in Section 4.6 and the scenario walk-through in Section 4.7 demonstrate the kernel–user-space partition under the eBPF verifier's execution constraints.

End-to-end data flow. The XDP program is attached to the network interface carrying GOOSE traffic between the publisher and the subscriber, where it observes every frame at the *Network Interface Card* (NIC) driver layer without interrupting forwarding. Each frame is parsed in two stages: first, the fixed-offset Ethernet, optional 802.1Q *Virtual Local Area Network* (VLAN) tag, and 8-byte GOOSE link-layer header (APPID, length, reserved fields) are read at constant offsets; second, the variable-length *Protocol Data Unit* (PDU) is decoded with a sequential ASN.1 *Basic Encoding Rules* (BER) walk that extracts the context-specific integer fields `stNum` and `sqNum`. The prototype limits in-kernel parsing to the fields required for its protocol-state checks. More extensive decoding of BER-encoded *Type-Length-Value* (TLV) fields complicates bounded iteration and variable-offset packet-access verification, so payload decoding is assigned to user space. Frames whose payload is needed for semantic analysis (in the physical-process context) are therefore copied to user space over a dedicated BPF ring buffer for secondary processing, while inline forwarding continues.

The fields extracted in the kernel are accumulated into a per-flow state record, held in a BPF hash map keyed on the directed `(src MAC, dst MAC)` tuple. A deterministic state machine compares each new packet against the most recent values stored in the corresponding flow entry, and any violation of retransmission invariants (for example, a backward `stNum` or a duplicate `sqNum`) is emitted as an anomaly record over a BPF ring buffer to a user-space consumer, whilst the frame itself is forwarded unmodified.

The user-space consumer aggregates the anomaly stream over a sliding window and writes deny entries keyed on the offending source *Medium Access Control* (MAC) into a second BPF hash map, which the XDP program consults on subsequent packets and against which it drops frames when the banned source recurs. This allows the data plane to act on deny-list entries written by user space after the initial detection decision.

Detection and Mitigation. The detection component evaluates each incoming GOOSE frame against the per-source baseline. Fast-path checks (counter monotonicity, inter-arrival bounds, and source legitimacy) execute entirely within the eBPF program; frames that exceed configured thresholds are forwarded to user space via the ring buffer for secondary assessment.

The mitigation component translates detection outcomes into per-source policy entries held in eBPF maps. The XDP program treats these entries as a match-action table, applying the corresponding action to each frame from the offending source, such as pass or drop.

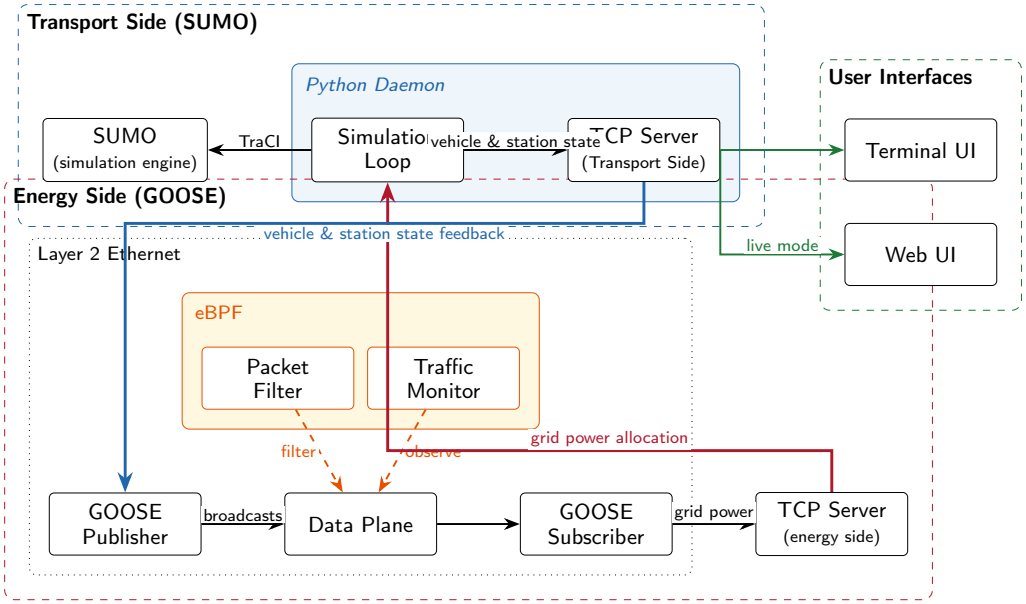


Figure 4.2: GOOSE data-plane analysis within the transport–energy co-simulation use case.

4.5.2 Use Case: Integration with Transport–Energy Co-Simulation

The use case integrates the GOOSE function with a bidirectional transport–energy co-simulation to illustrate how an application can exchange external state alongside protocol analysis. In the integration prototype, a Co-simulation Client ingests transport-side state, and a State Exporter returns energy-side supply state.

The use case is motivated by the relationship between EV charging and substation operation (Al-Rubaye *et al.*, 2019; Fink *et al.*, 2025; Pittalà *et al.*, 2024). As road transport electrifies, charging demand couples transport-system behaviour to the distribution feeder served by the substation, making the transport–energy nexus a relevant setting for exploring external context.

Figure 4.2 places the GOOSE analysis function and its user-space policy support on the Energy Side within the wider co-simulation environment. The transport simulator, state-exchange channels, and observability interfaces support this integration use case. The architecture is partitioned into two co-simulation domains, each managed by a dedicated Python daemon, coupled to each other over TCP.

On the Transport Side, *Simulation of Urban MObility* (SUMO) models urban traffic and EV charging behaviour; the daemon's simulation loop queries SUMO via *Traffic Control Interface* (TraCI) and forwards vehicle and station state to the Energy Side through a TCP interface.

On the Energy Side, GOOSE frames transit a Layer-2 Ethernet data plane between a software publisher and subscriber; two eBPF components, Packet

Filter and Traffic Monitor, operate on this data plane to enforce filtering rules and record traffic statistics. Anomaly records and bounded packet copies are exported asynchronously to user space. A second TCP channel carries grid power allocation from the energy side back to the Simulation Loop, closing the cross-domain coupling: EV charging demand shapes the transport simulation, and the resulting mobility state informs the context available to the anomaly-assessment logic.

The information flow between the two domains reflects how deployed EV charging infrastructure is already interconnected with, and remotely managed in relation to the power grid at scale (Nasr *et al.*, 2023; Rivera *et al.*, 2022). Distribution-grid operators increasingly coordinate EV charging through a *Distributed Energy Resource Management System* (DERMS), and aggregated EV charging is itself treated as a controllable element of grid energy storage that participates in dynamic energy trading, so that charging demand can be dispatched to absorb or relieve grid stress (C. Jin *et al.*, 2013). The transport-to-energy coupling modelled here therefore demonstrates that the mobility and charging state supplied by the transport side is the same class of signal a DERMS would consume to allocate grid power, and the power-allocation feedback returned to the simulation loop mirrors the dispatch decision such a system would issue (P. Bauer *et al.*, 2025).

The implementation simplifies this interface as follows: both directions are carried over plain TCP channels exchanging aggregate state variables, and the model omits the low-level details such as middleware, authentication, transport-layer timing, and partial-failure behaviour that a production DERMS integration would exhibit.

The remaining components in Figure 4.2, a Terminal *User Interface* (UI) and a Web UI, increase the observability of the system: both expose live state from the two domains for debugging and demonstration, and the Web UI additionally records sessions for post-hoc replay and analysis of attack scenarios.

The implemented coupling supplies cross-domain state and a simplified charging response.

4.6 Implementation

4.6.1 Prototype Environment

The GOOSE fabric is implemented on a commodity x86-64 platform running Debian 13 (trixie) with Linux kernel 6.x, which provides native support for eBPF/XDP. The GOOSE environment is built using the libiec61850 library v1.6.1 (MZ Automation GmbH, 2024), which facilitates the generation and parsing of IEC 61850 GOOSE frames. The eBPF toolchain comprises clang

19.1.7 as the BPF-target compiler, libbpf 1.5.0 for runtime program loading and ring-buffer consumption, and bpftool 7.5 for runtime introspection and BPF map manipulation by the user-space control plane.

The substation network is emulated on the same host using four Linux network namespaces: `ied_grid0`, `ied_grid1`, `observer` and `subscriber` are connected via *Virtual Ethernet* (veth) pairs to a single Linux bridge (`br_substation`) that carries the GOOSE multicast traffic between the IED publishers and the subscriber. The bridge floods GOOSE traffic to every port as it would on a substation switch; each namespace is given a distinct, locally-administered IED MAC address so that the data plane observes every legitimate publisher under a stable identity, which is the form in which the per-flow state map keys its entries.

The transport-side co-simulation environment is built on SUMO v1.24.0, using TraCI to connect the Python daemon to the simulator and exchange state in real time.

To establish ground truth for cross-validation, the same captured GOOSE frames are processed in parallel by the Wireshark command-line front-end (`tshark`), and the parsed field dumps from the in-kernel parser, the user-space parser, and `tshark` are diffed in an automated step that runs as part of the validation.

4.6.2 Experimental Data and Transmission Conditions

In the co-simulation setup, Python publishers obtain charging-demand state from SUMO and calculate rated and output power using the simplified energy model. These values are passed to libiec61850 publishers to generate GOOSE frames.

The detector experiments use two open data sets: PowerDuck normal-traffic captures for the packet-latency benchmark (Zemanek *et al.*, 2022), and QUT-ZSS-2023-GOOSE attack captures for the mitigation tests.[1] For testing in the local setup, captured frames were adapted by modifying their MAC addresses and payload values.

Scapy is used to read and replay the captured traffic. The QUT replay utility selects GOOSE frames and removes VLAN tags. Packet latency is measured over a veth pair using `tcpdump`; mitigation tests use loopback with generic XDP and a sender pinned to one *Central Processing Unit* (CPU). The latency replay uses a configured 1 ms inter-packet wait, while the mitigation replay disables waits based on capture timestamps.

[1] Data source: https://github.com/CSCRC-SCREED/QUT-ZSS-2023-GOOSE.

Violation	Condition
ST_BACKWARD	stNum < stored stNum
SQ_BACKWARD	sqNum < stored sqNum
SQ_DUPLICATE	sqNum == stored sqNum
ST_JUMP	stNum > stored stNum + 1
SQ_JUMP	sqNum > stored sqNum + 1

Table 4.1: Sequence-number violation conditions for GOOSE frames.

4.6.3 In-Kernel Fast-Path Detection

The fast-path checks motivated in Section 4.2 are realised entirely within the end-host kernel data plane: a single XDP program attached at the NIC driver layer consults state held in BPF maps before frames traverse the kernel network stack, the end-host programmable counterpart to the in-switch P4 data plane studied in Chapter 3. Locating the per-packet checks in-kernel is what allows the IEC 61850 protection budget to be met without a user-space round-trip. The deep packet inspection that exceeds the eBPF verifier's parsing reach is partitioned to user space.

The program (`goose_xdp.c`) is attached to the interface that carries GOOSE multicast traffic between the publisher and subscriber namespaces; it intercepts every frame and, absent a deny-list match on the source MAC, returns `XDP_PASS` so that detection runs transparently without interrupting the forwarding decision.

A deterministic state machine enumerates mutually exclusive sequence-number violations of the IEC 61850-8-1 retransmission invariants as follows:

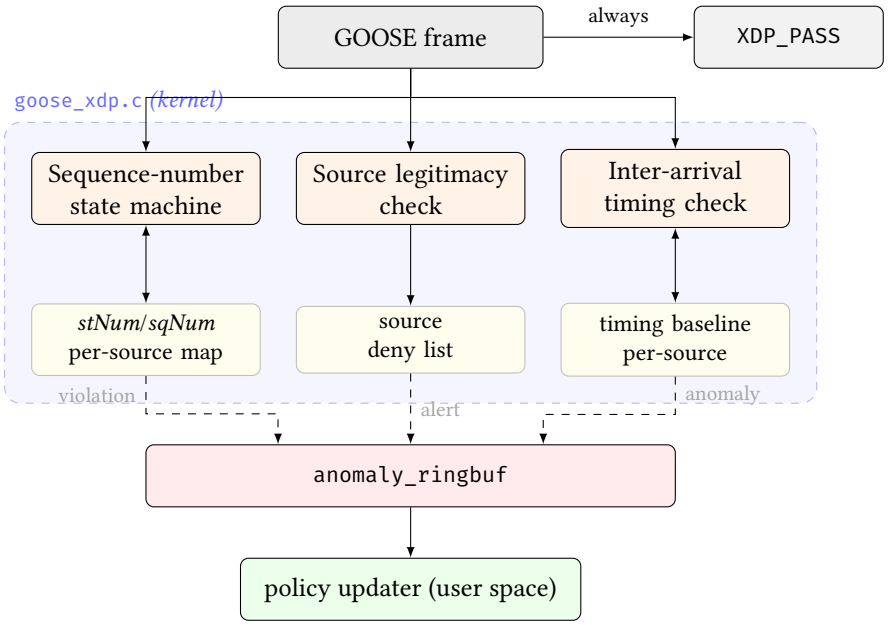


Figure 4.3: In-kernel GOOSE anomaly-detection pipeline.

Except for the sequence-number related violations (Figure 4.3), the other two detectors are independent of the state machine: the source legitimacy check is a simple membership test against the per-source deny list, and the inter-arrival timing check compares the current packet's timestamp against the distribution of inter-arrival times observed from that source during the discovery phase.

Moreover, as there is no trust assumption on the source, the program only implements a deny list and does not attempt to maintain a per-source allow list, which would be vulnerable to poisoning by an adversary who can inject frames from a spoofed source. Instead, the program treats any source that violates the sequence-number invariants as anomalous and emits an alert to user space, where the policy updater can decide how to respond based on the broader context.

The data plane exposes its results to user space through three channels with distinct latency–fidelity trade-offs:

- `flows` (`BPF_MAP_TYPE_PERCPU_HASH`, capacity 4096) keyed on the directed `(src MAC, dst MAC)` tuple, holding per-flow state and statistics that the user-space consumer can poll at low frequency for slowly-updating dashboards. The per-CPU choice trades cross-CPU state coherence for lock-free updates.
- `anomaly_ringbuf` (`BPF_MAP_TYPE_RINGBUF`, 256 kB) carrying a 48-byte record per detected anomaly, providing sub-millisecond push delivery to a libbpf-based consumer.
- `pkt_ringbuf` (`BPF_MAP_TYPE_RINGBUF`, 1 MB) carrying a copy of the leading 64–192 bytes of every GOOSE frame, with the copy length selected at compile time by a fixed-length ladder of `bpf_xdp_load_bytes` calls because the verifier requires constant-width arguments to packet-access helpers; this third channel is what the user-space deep parser consumes for the BER fields the kernel cannot extract (Deokar *et al.*, 2024).

4.6.4 User-Space Orchestration

Building on the module decomposition of Section 4.5, this subsection details how each component of the user-space slow path is realised (Figure 4.4). The partitioning instantiates the shared-ownership arrangement of Section 2.5: the per-packet checks expressible within verifier limits (header-counter monotonicity, inter-arrival timing, and *Access Control List* (ACL) lookup) and the resulting pass/drop decision remain in the kernel fast path, whilst user space hosts the deep parser for the BER fields the kernel cannot extract and updates enforcement policy from anomaly events streamed by the kernel detector. The co-simulation client ingests external state through the

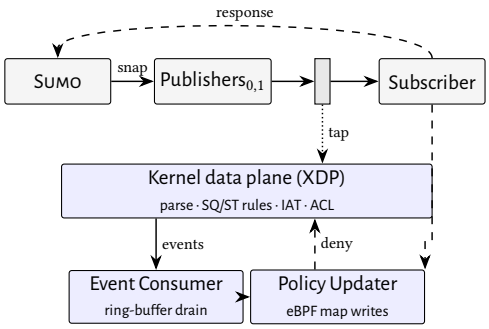


Figure 4.4: GOOSE detector orchestration and co-simulation interface.

physical-context interface. The following paragraphs describe kernel-event ingestion, the co-simulation context interface, and the supervisor that orders their lifecycle.

Kernel-to-user-space ingestion. Similar to the pattern of kernel–user-space partitioning in the VNF model of Van Tu *et al.* (2019), the in-kernel detector emits anomaly events and packet copies to user space through BPF ring buffers, which the event consumer drains through the `libbpf` polling interface (Figure 4.4).

The choice of ring buffer over the older perf buffer is motivated by the multi-producer, single-consumer ordering and reservation semantics that ring buffers expose, which preserve the temporal order of suspect events emitted by the detection module along the GOOSE fast path (C. Liu *et al.*, 2024).

External-state ingestion via co-simulation. The co-simulation client connects to the Python-based process daemon of the SUMO instance and polls vehicle-level and aggregate-load state in *JavaScript Object Notation* (JSON) format aligned with the simulation step. A polling discipline is preferred over a push-based one because TraCI is fundamentally a request–response protocol, and, more importantly, because the orchestrator must remain authoritative over the timing at which physical-world context would be folded into in-network policy: the refresh rate is itself a security-relevant parameter, and exposing it as a decision surface that the simulator could drive would compromise the integrity of the detection pipeline. Polled state is staged in a versioned in-memory snapshot that a decision module can read atomically; consumer threads never block on the TraCI socket, so transient simulator stalls degrade context freshness while detection remains available.

Lifecycle and orchestration. The supervisor enforces a deterministic startup sequence: it loads the eBPF object file, pins the maps required for cross-component visibility, attaches the XDP program to the configured interface, starts the event consumer and policy updater threads, and only then opens the transport connection.

This ordering starts the detector and its policy support before the co-simulation connection. Shutdown reverses the order under a fail-stop discipline: the simulation connection is closed first, the consumer threads then drain the ring buffer to a quiescent state, and finally the XDP program is detached and the maps are unpinned. Internal liveness is surfaced through a control socket, which is used by the testbed harness of Section 4.6.1 to coordinate experimental runs.

4.7 Attack Walk-Throughs

4.7.1 Denial of Service

To launch a *Denial of Service* (DoS) attack against the GOOSE data plane, an adversary with access to the substation LAN can either (a) flood the network with a high volume of GOOSE frames to overwhelm the subscriber's processing capacity, or (b) craft frames with large `stNum`/`sqNum` jumps, which subscriber IEDs may treat as authoritative and consequently suppress subsequent legitimate frames.

In scenario (a), the inter-arrival timing check raises an alert when the observed arrival rate from a single source exceeds the per-source baseline, but the XDP program still passes all frames (`XDP_PASS`) to avoid inadvertent disruption of legitimate traffic. Enforcing a rate limit would add a further match-action rule to the deny-list map, a direct extension along the same deny-list mechanism, not exercised in the current prototype.

In scenario (b), the injected frames trigger the `SQ_JUMP` and `ST_JUMP` conditions of the sequence-number state machine (Table 4.1). Within the XDP program, the per-flow state record is updated and an anomaly record is pushed to `anomaly_ringbuf`. The user-space event consumer aggregates the violation stream over a configurable sliding window; once the count exceeds the per-source threshold, the policy updater writes a deny-list entry keyed on the offending source MAC into the eBPF deny-list map. Subsequent frames from that source match the source-legitimacy check (Figure 4.3) and are dropped by the XDP program before they can reach the subscriber, thereby terminating the attack traffic at an early stage.

4.7.2 False Data Injection

An adversary aiming to inject false measurement data into the substation constructs a GOOSE frame whose sequence counters (`stNum`, `sqNum`) are syntactically valid increments of the observed sequence, but whose DataSet payload carries a spoofed measurement, for example a false overcurrent reading that would cause the subscriber IED to assert an unexpected trip command.

Because the counter values satisfy the sequence-number invariants, the state machine in the XDP program does not flag the frame. However, the XDP program copies the GOOSE frame to `pkt_ringbuf` for secondary processing; the user-space deep parser would extract the DataSet fields from the BER-TLV payload and compare the injected values against the co-simulation snapshot maintained. If the injected measurement mismatches the digital-twin state, the policy updater would classify the source as anomalous and write a deny-list entry to the eBPF map (Mahboob *et al.*, 2026).

This walk-through identifies a limit of the implemented function: counter-based checks alone cannot distinguish an attack that preserves the expected sequence. Comparing payload values with digital-twin state is a proposed use of the integration interface.

4.7.3 Replay Attack

An adversary captures a legitimate GOOSE trip-command frame during a real fault event, when `stNum` is at some value s. After the fault has been cleared and the publisher has resumed normal heartbeat transmissions (advancing `stNum` to $s + k$ for some $k \geq 1$), the adversary replays the captured frame.

The replayed frame carries `stNum` $= s$, which is strictly less than the stored value $s + k$ in the per-flow state record held by the XDP program. This triggers the `ST_BACKWARD` violation condition (Table 4.1), causing the XDP program to emit an anomaly record to `anomaly_ringbuf`. The event consumer receives the record and the policy updater writes a deny-list entry for the replaying source MAC.

A more sophisticated adversary may time the replay to coincide with a legitimate `stNum` increment, so that the replayed value is not immediately backward. In this case, the `sqNum` regression provides the discriminating signal: the replayed frame's `sqNum` restarts at zero (as it was when captured), whilst the live stream has advanced to a higher retransmission offset, triggering the `SQ_BACKWARD` condition.

4.8 Architectural Comparison and Indicative Measurements

What matters for an industrial or edge deployment is not the absolute latency attainable on a particular target, but whether the in-kernel detection path stays within the real-time budget that the substation network imposes on GOOSE, a millisecond-scale requirement defined by IEC 61850. The measurements characterise packet-processing overhead and policy reaction under the specified host and replay conditions.

All measurements in this section were taken on a single Debian 13 (Linux 6.12.33)

Table 4.2: BPF map allocations of the proposed in-kernel GOOSE detection.

Map	Type	Entries	Per entry	Footprint
`flows`	`PERCPU_HASH`	4096	104 B	~1.6 MB
`anomaly_ringbuf`	`RINGBUF`	–	–	256 KB
`pkt_ringbuf`	`RINGBUF`	–	–	1 MB

host equipped with an Intel N100 (4 cores@3.4 GHz boost) and 16 GB of *Random Access Memory* (RAM), with clang 19.1.7 and libbpf 1.5 providing the eBPF toolchain.

Before turning to those figures, this section situates the in-kernel component of the proposed prototype against SGFabric (Holik, 2024), which targets the same class of substation traffic and therefore provides a natural *architectural* reference point. SGFabric is realised as a network function over BPFabric (Jouet & Pezaros, 2017), a framework that accepts eBPF bytecode as the specification of data-plane behaviour but executes it in user space.

Placed side by side, the two designs share an eBPF bytecode programming model but differ in execution context: SGFabric runs the bytecode in a user-space runtime, whereas the proposed prototype runs it on the in-kernel XDP fast path. SGFabric provides the architectural comparison; the latency measurements compare the GOOSE XDP function with a no-op baseline.

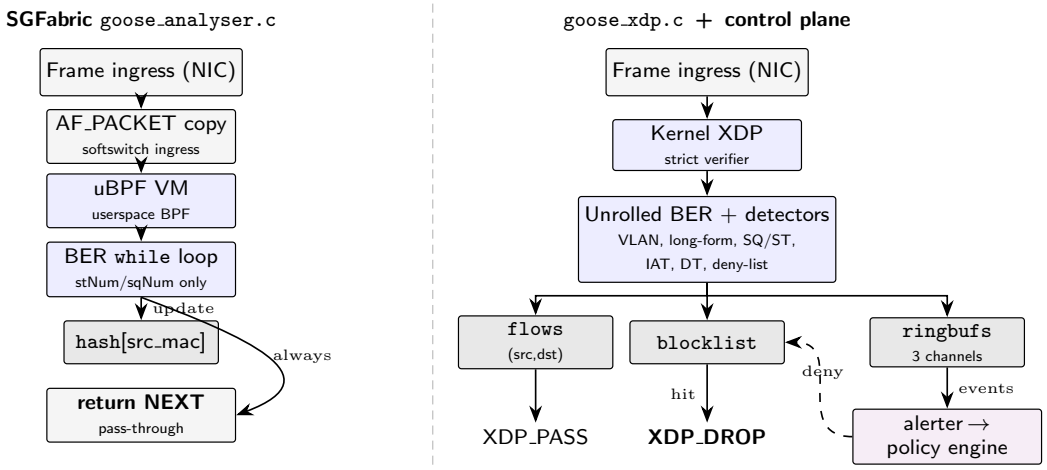


Figure 4.5: XDP and SGFabric GOOSE detection pipelines.

Figure 4.5 illustrates the pipeline structures of SGFabric's GOOSE analyser and the proposed in-kernel XDP program. The pinned-memory footprint of the XDP program is shown in Table 4.2. Two ring buffers, `anomaly_ringbuf` and `pkt_ringbuf`, are provisioned to carry the anomaly events and packet clones out of kernel space. These circular byte buffers hold variable-length records and are allocated as a single region of a chosen total size; the *Entries* and *Per entry* columns therefore do not apply to them (a dash in the table), and their footprint is simply the configured capacity, here 256 KB and 1 MB

respectively. The `flows` map is provisioned to hold per-flow state and statistics for detection and user-space monitoring purposes, with a capacity of 4096 entries that can be adjusted based on the expected number of concurrent flows in the deployment environment. Each entry takes 104 bytes, including padding for alignment. Because `PERCPU_HASH` allocates one copy of the table per CPU, the total footprint scales linearly with the host's core count: on the four-core testbed described above it amounts to approximately 1.6 MB, and a 32-core host would consume roughly eight times more for the same logical capacity.

On the left, the SGFabric program runs inside the user-space BPFabric runtime: each frame is first lifted out of the kernel through an `AF_PACKET` copy into the BPFabric softswitch and dispatched to a uBPF virtual machine, where a BER `while` loop extracts `stNum` and `sqNum` and writes them into a hash map keyed on the source MAC address; the program then returns `NEXT` so that the frame proceeds to the next stage of the network-function chain. It is a modular design that could be extended with additional stages.

On the right, the XDP program sits in the kernel XDP hook under the strict verifier, where an unrolled BER parser extracts the same fields and additionally maintains per-flow state and performs the sequence-number and inter-arrival checks within the same execution context. The program returns `XDP_PASS` for every frame except those whose source MAC hits the deny-list, so that the detection logic operates transparently without interrupting the forwarding decision, and emits records to user space through two dedicated ring-buffer channels for secondary processing by the event consumer and policy updater.

To quantify the additional delay introduced by the XDP-side checks, the detection path was measured on the PowerDuck (Zemanek *et al.*, 2022) trace against a plain XDP program that returns `XDP_PASS` immediately. End-to-end latencies are obtained from `tcpdump` traces on the veth interfaces across five runs of 5000 frames each.

Two XDP programs are loaded in turn on the receive-side of a single veth pair `veth_a <--> veth_b`:

- **No-op XDP:** a minimal XDP program that pays the full XDP framework overhead but performs no processing on the packet, simply returning `XDP_PASS` for every frame.
- **GOOSE XDP:** the proposed in-kernel detection program.

Table 4.3 shows that, at the median, the parser raises the end-to-end per-packet latency from 2766 ns to 3815 ns, an increase of 1049 ns over the no-op baseline, but still below the GOOSE re-transmission threshold ($\leq 3ms$ by IEC 61850); the kernel-internal counter reports a comparable overhead of

Table 4.3: Per-packet latency (ns) of `goose_xdp` vs. a no-op XDP baseline.

Configuration	Mean	p50	p95	p99
No-op XDP (`return XDP_PASS`)	3313	2766	6533	10824
`goose_xdp`	4949	3815	9250	14448

Kernel-internal XDP runtime (`bpf_stats`): no-op 80 ns, `goose_xdp` 1313 ns per packet ($\Delta_{\text{parser}} = 1233$ ns).

1233 ns per packet. The measured processing overhead is small relative to the millisecond-scale protection budget on this host and workload.

Beyond this per-packet cost, the prototype was exercised end to end to examine how quickly the detect-then-deny loop reaches enforcement once an attack begins. Table 4.4 demonstrates the outcome for the two attack classes whose `stNum`/`sqNum` violations accumulate past the policy threshold ($R = 3$); the five payload-only false-data-injection classes of Section 4.7.2 leave the sequence counters intact and never cross it. The reaction interval t_{react}, measured from the threshold-triggering anomaly to the first subsequently blocked frame using kernel event timestamps, is 127 ± 21 ms for `high_stnum` and 116 ± 13 ms for `flood_del` (mean $\pm$ 95 % confidence interval over five single-core runs), and includes event delivery, the user-space map update, and the wait for the next frame from that source. These values characterise the single-core replay conditions in Section 4.6.2. The accumulation term $t_{\text{accumulate}}$ depends on the attacker's replay pace and is computed offline from the recorded trace. The reported t_{total} is accordingly a derived sum of two terms obtained in separate replay realisations, and provides an indicative time-to-first-enforcement estimate.

Table 4.4: Indicative mitigation timings for attacks crossing the replay-class threshold ($R = 3$).

Attack class	$t_{\text{accumulate}}$ (ms)	t_{react} (ms)	t_{total} (ms)
`high_stnum`	2006	127 ± 21	≈ 2133
`flood_del`	343	116 ± 13	≈ 459

4.9 Discussion and Summary

This chapter developed and measured a stateful GOOSE analysis function with in-kernel checks and user-space policy maintenance, and presented digital-twin co-simulation integration as a use case. The prototype and three attack walk-throughs illustrate its processing paths, while Tables 4.3 and 4.4 characterise per-packet overhead and policy-update latency.

4.9.1 State Ownership and Function Partitioning

The two systems divide maintenance responsibilities differently. The DNS cache of Chapter 3 maintains cache and epoch state within the data plane, including lookup, invalidation, and refresh. In the GOOSE prototype, the XDP program maintains sequence state, while user space owns anomaly aggregation and deny-list updates.

State ownership should be assigned according to the operation that maintains it. Per-packet sequence updates belong with the checks that consume them, while aggregation and enforcement-policy changes can be maintained separately. This makes the coordination boundary explicit without requiring every stateful operation to execute in the data plane. The prototype also shows how verifier constraints shape the parsing boundary: extracting the fields needed for bounded checks allows more extensive decoding to proceed asynchronously in user space.

4.9.2 Data-Plane Independence and Enforcement Delay

This division addresses RQ2: bounded sequence and timing checks remain in the kernel, events pass asynchronously through ring buffers, and user-space policy updates become available to subsequent packets through shared BPF maps. The fast path does not wait for the event consumer or policy updater, although enforcement of a newly detected anomaly is delayed by aggregation and map updates. The measurements in Section 4.8 distinguish microsecond-scale packet processing from roughly hundred-millisecond policy reaction; reporting only the former would conceal the delay before subsequent packets are denied.

4.9.3 External Context and Policy Design

The physical-context interface provides a basis for future context-aware detection. The false-data-injection example shows why protocol-consistent counters cannot establish payload legitimacy. For other stateful functions, external context should therefore be treated as a separate policy input, with its freshness and effect on decisions assessed using an explicit decision rule and labelled evaluation data.

4.9.4 Scope and Remaining Limitations

Building on the architecture and prototype developed above, an extension would naturally focus on the following directions:

- Measuring throughput and evaluating deployment at scale.

- Coupling the testbed with a power system model, so that the in-kernel detector can be exercised under production-scale GOOSE traffic in a hardware-in-the-loop configuration.
- Broadening the `stNum`/`sqNum` baseline, presently learned from a clean simulation trace with each source MAC treated as a distinct flow, to baselines trained across heterogeneous publishers (vendors) and operating conditions (with different burst patterns).
- Once physical-process context is folded into detection policy, the user-space polling that would refresh that context into a BPF policy map would introduce staleness determined by simulation progress, polling intervals, and communication delay; during rapid EV charging transitions this staleness could exceed the GOOSE retransmission window, so that a freshly retransmitted frame would be assessed against an out-of-date load context.

The GOOSE prototype demonstrates how user-space policy maintenance can support an independent packet-processing fast path. These coordination costs and remaining context-integration requirements inform the application development guidelines in Chapter 5.

Chapter 5

Guidelines for Developing Stateless and Stateful Applications

This chapter addresses RQ3 by synthesising the development lessons of the *Domain Name System* (DNS) cache (Chapter 3) and *Generic Object Oriented Substation Event* (GOOSE) detector (Chapter 4). It derives guidelines for stateless and stateful applications using the framework of state ownership and data-plane independence introduced in Section 2.5.

Stateless operations need no mutable cross-packet history, while stateful operations maintain information across packets. Both systems maintain persistent state under different ownership arrangements. Their lessons concerning bounded processing apply to both application classes, while their maintenance and coordination requirements additionally inform stateful design.

5.1 Lessons Learnt from the Two Systems

5.1.1 Jointly Design State Representation and Packet Processing

The DNS cache shows that storage and processing choices also determine service behaviour: bounded probing limits admission, compact fingerprints introduce possible false matches, and shared epochs with passive refresh change freshness guarantees (Section 3.5).

5.1.2 Separate Immediate Decisions from Policy Maintenance

The GOOSE detector shows that an independent fast path can use software-maintained policy when updates take effect asynchronously (Sections 4.5

and 4.6). This arrangement requires separate timing budgets for packet processing and enforcement: fast checks do not eliminate the delay before a newly detected condition changes policy (Section 4.8). Future use of the co-simulation context interface would additionally require assessing context freshness and its effect on policy decisions.

5.1.3 Distinguish Execution Bounds from Measured Performance

P4 processing must fit the target's parser, field widths, and pipeline resources; *Extended Berkeley Packet Filter* (eBPF) packet accesses and control flow must satisfy the verifier. Both encourage bounded operations, whose performance depends on the target and workload. The DNS evaluation uses *Behavioral Model v2* (BMv2) and a controlled cache workload (Section 3.7.2), while the GOOSE measurements use a host replay testbed (Section 4.8). This comparison motivates measuring the cost of the chosen operations under the application's intended conditions.

5.2 Guidelines for Application Development

For stateless applications, bound parsing and computation, define supported inputs and fallback behaviour, and assess whether each processing path can complete independently. Configured-rule updates should be coordinated without introducing unnecessary per-packet software dependencies.

For stateful applications, assign responsibility for state creation, update, invalidation, and refresh. Select data-plane ownership when these operations fit the target and their admission, accuracy, and freshness trade-offs satisfy service requirements.

Where software capabilities are needed, use asynchronous maintenance if packets can be handled with the currently installed state and the service tolerates delayed updates. Define the coordination interface and acceptable delay before updates affect subsequent packets. If a packet requires a new software decision before forwarding, account for that synchronous dependency in its latency budget.

Together, these guidelines answer RQ3 by connecting state ownership and data-plane independence to service requirements, resource costs, and coordination obligations. Chapter 6 revisits their implications for the thesis statement.

Chapter 6

Conclusion

This thesis examined two state-maintenance arrangements. Chapter 3 developed a P4 *Domain Name System* (DNS) cache with data-plane-owned runtime state. Chapter 4 developed a *Generic Object Oriented Substation Event* (GOOSE) detector with kernel checks and user-space policy maintenance. Chapter 5 synthesised lessons from both systems to develop application guidelines and answer RQ3.

This concluding chapter revisits the thesis statement in light of those findings, identifies future research directions, and closes with broader remarks on programmable data planes.

6.1 Thesis Statement Revisited

This thesis addressed three successive research questions concerning the scope and organisation of in-network offloading on programmable data planes: the costs and restrictions of self-contained processing, the extension to software-coordinated processing, and the implications of both for offloading decisions.

RQ1 asked what resource and processing costs, and what restrictions on service behaviour, arise when a network service that conventionally requires control-plane intervention is realised as a fully self-contained data-plane pipeline. The data-plane-maintained DNS cache (Chapter 3) answers this for network functions that decompose into fixed-width key-value operations: by jointly tailoring storage structure, parser, collision scheme, eviction policy, and update path to the constraints of a P4 pipeline, a practical caching service can be sustained entirely within the data plane without any per-packet control-plane intervention; the costs include additional pipeline complexity and register accesses, bounded cache admission, and the loss of per-entry control-plane metadata such as *Time-to-Live* (TTL) counters and access timestamps. These choices restrict cache behaviour, as synthesised in Sections 5.1.1 and 5.2. This provides evidence for the first claim of the thesis statement (*so*).

RQ2 asked how processing and state maintenance can be partitioned between the data plane and end-host software when service requirements exceed the constraints of a self-contained pipeline, while retaining an independent packet-processing fast path. The GOOSE anomaly detection architecture (Chapter 4) addresses this by partitioning detection logic across kernel space and user space: lightweight sequence-number validation runs in *eXpress Data Path* (XDP) on the fast path, event aggregation and policy management run in user space, and a co-simulation channel provides an interface for future context-aware analysis. This architecture provides evidence for the second claim (*S1*), illustrating how user-space coordination can extend in-kernel processing through modular layering within an *Extended Berkeley Packet Filter* (eBPF)/XDP stack.

RQ3 asked how state ownership and data-plane independence, together with their associated costs and coordination requirements, inform network-function offloading decisions. The guidelines in Chapter 5 support the third claim (*S2*) by relating the resource costs, service restrictions, and coordination requirements observed in the two systems to the selection of an offloading arrangement for stateless and stateful applications.

Both systems expose the constraints that shape the division of runtime state-maintenance responsibilities between the data plane and end-host software, as described by the framework in Chapter 2. Their cross-cutting implications are developed in Chapter 5; the following sections consider the broader research directions they motivate.

6.2 Future Directions

A long-term vision emerging from this thesis is to develop programmable data planes as transparent middleware within controlled network domains, preserving application interfaces and end-to-end service semantics. The distinction between state-maintenance responsibilities and per-packet processing dependencies motivates two research directions: packet-carried context and selective processing.

6.2.1 Packet-Carried Context Within Programmable Domains

A larger internal *Maximum Transmission Unit* (MTU), potentially using 9000-byte jumbo frames, could accommodate processing context beyond the conventional 1500-byte budget while leaving endpoint packet sizes unchanged. Domain-local headers could carry operation identifiers or state versions, with encapsulation added at ingress and removed at egress, as exemplified by *In Situ Operations, Administration, and Maintenance* (IOAM) (Brockners *et al.*, 2023). Larger MTUs nevertheless do not remove target-specific parsing and execution limits. The research challenge is to determine what con-

text should travel with packets and what state should remain at execution sites, allowing internal capabilities to evolve without exposing new packet formats to applications.

6.2.2 Selective In-Network Processing With a Placement Advantage

A design principle is to place processing in the network where its position provides a demonstrable benefit that justifies resource and coordination costs, while retaining endpoint responsibility for complete correctness (Saltzer *et al.*, 1984). AllReduce aggregation for distributed *Artificial Intelligence* (AI) training illustrates this opportunity: combining contributions inside the network can reduce subsequent transmission, as demonstrated by SwitchML's joint design of switch and end-host processing (Sapio *et al.*, 2021). A long-term goal is to derive placement criteria that balance communication savings against state synchronisation costs, additional failure dependencies, recovery overhead, and forwarding resource consumption, guiding which operations belong in switches, host-attached accelerators, or endpoint software. These criteria should also specify how processing can be bypassed or withdrawn without changing application interfaces or compromising end-to-end correctness.

6.3 Concluding Remarks

Network programmability is at an inflection point. It is becoming less a property of a particular device and more a property of the entire path a packet traverses. As that shift unfolds, the unit of design migrates from *which device implements a function* to *how a function is decomposed and distributed across devices whose execution models might differ*.

The two systems show how this design question can be examined through state representation, execution-model constraints, and the cost of crossing between layers.

The broader vision is that of a future network in which the data plane is not a monolithic entity but a heterogeneous collection of execution sites, each with its own execution model and state structures, spanning the per-packet processing path from the network fabric through the kernel to user space.

References

Abranches, M., Michel, O., & Keller, E. (2022). Getting back what was lost in the era of high-speed software packet processing. *Proceedings of the 21st ACM Workshop on Hot Topics in Networks*, 228–234 (see p. 2).

Agrawal, A., & Kim, C. (2020). Intel tofino2–a 12.9 tbps p4-programmable ethernet switch. *2020 IEEE Hot Chips 32 Symposium (HCS)*, 1–32 (see pp. 2, 18–20).

Ahad, A., Bakar, R. A., Arslan, M., & Ali, M. H. (2023). DPIDNS: A Deep Packet Inspection Based IPS for Security Of P4 Network Data Plane. *2023 International Conference on Smart Computing and Application (ICSCA)*, 1–8 (see pp. 24, 27).

Akem, A. T.-J., Gucciardo, M., & Fiore, M. (2023). Flowrest: Practical flow-level inference in programmable switches with random forests. *IEEE INFOCOM 2023-IEEE Conference on Computer Communications*, 1–10 (see pp. 14, 19).

Al-Rubaye, S., Kadhum, E., Ni, Q., & Anpalagan, A. (2019). Industrial Internet of Things Driven by SDN Platform for Smart Grid Resiliency. *IEEE Internet of Things Journal*, *6*(1), 267–277. https://doi.org/10.1109/JIOT.2017.2734903 (see p. 61).

AlSabeh, A., Kfoury, E., Crichigno, J., & Bou-Harb, E. (2022). P4ddpi: Securing p4-programmable data plane networks via dns deep packet inspection. *NDSS Symposium 2022* (see pp. 24, 27).

Amin, R., Reisslein, M., & Shah, N. (2018). Hybrid SDN networks: A survey of existing approaches. *IEEE Communications Surveys & Tutorials*, *20*(4), 3259–3306 (see p. 2).

Arges, C. J. (2021, December 6). *How We Used eBPF to Build Programmable Packet Filtering in Magic Firewall*. Cloudflare. Retrieved October 12, 2026, from https://blog.cloudflare.com/programmable-packet-filtering-with-magic-firewall/ (see p. 15).

Bauer, L., Beckers, T., Sweet, N., dependabot[bot], Lehner, F., Meleleo, M., Gögge, R., Klauser, T., Cacheux, P., Reimerink, D., Kahle, B., olsajiri, Fabre, A., Rad, M. A., Ylinen, M., Falzoi, F., Böhm, T., eiffel-fl, Zavaritsky, N., … Li, H. (2025, November). cilium/ebpf. https://github.com/cilium/ebpf (see p. 16).

Bauer, P., i Carbonell, J. R., Wilker, S., & Sauter, T. (2025). A SUMO-Simulated Reinforcement Learning Framework for Bidirectional

Charging of Electric Vehicles. *2025 IEEE International Conference on Industrial Technology (ICIT)*, 1–7 (see p. 62).

Ben Basat, R., Ramanathan, S., Li, Y., Antichi, G., Yu, M., & Mitzenmacher, M. (2020). PINT: Probabilistic in-band network telemetry. *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 662–680 (see pp. 24, 27).

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., *et al.* (2014). P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, *44*(3), 87–95 (see pp. 1, 2, 9, 17).

Bosshart, P., Gibb, G., Kim, H.-S., Varghese, G., McKeown, N., Izzard, M., Mujica, F., & Horowitz, M. (2013). Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. *ACM SIGCOMM Computer Communication Review*, *43*(4), 99–110 (see pp. 2, 9, 12, 14).

Brockners, F., Bhandari, S., Bernier, D., & Mizrahi, T. (2023, April). In Situ Operations, Administration, and Maintenance (IOAM) Deployment. https://doi.org/10.17487/RFC9378 (see p. 78).

Chen, X., Wu, C., Liu, X., Huang, Q., Zhang, D., Zhou, H., Yang, Q., & Khan, M. K. (2023). Empowering network security with programmable switches: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, *25*(3), 1653–1704 (see p. 56).

Chen, X., Zhang, W., Sun, X., Liu, H., Zhang, J., Huang, Q., Zhang, D., Zhou, H., Liu, X., & Wu, C. (2024). Resource-Efficient and Timely Packet Header Vector (PHV) Encoding on Programmable Switches. *IEEE/ACM Transactions on Networking*, *32*(5), 4191–4206 (see pp. 14, 22).

Choueiri, S., Mazloum, A., Kfoury, E. F., & Crichigno, J. (2024). Scalable heavy hitter detection: A DPDK-based software approach with P4 integration. *GLOBECOM 2024-2024 IEEE Global Communications Conference*, 1930–1935 (see pp. 25, 28).

Cilium. (2026). BPF and XDP Reference Guide: BPF Architecture [Accessed June 2026]. https://docs.cilium.io/en/stable/reference-guides/bpf/architecture/ (see p. 17).

Cisco Popularity List [https://umbrella-static.s3-us-west-1.amazonaws.com/index.html, Accessed June 1 2024]. (2016). (See pp. 32, 47, 49).

Cloudflare. (2026). Cloudflare Radar: Domain Rankings Worldwide [Accessed: 2026-04-20]. https://radar.cloudflare.com/domains (see p. 32).

Dai, Y., Liu, G., Huang, T., Wang, S., Wang, Y., Wu, X., Song, L., Li, H., Wang, C., Hu, F., *et al.* (2025). DNSLogzip: A Novel Approach to

Fast and High-Ratio Compression for DNS Logs. *Proceedings of the ACM SIGCOMM 2025 Conference*, 467–480 (see p. 33).

Deokar, M., Men, J., Castanheira, L., Bhardwaj, A., & Benson, T. A. (2024). An empirical study on the challenges of ebpf application development. *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, 1–8 (see p. 65).

Ding, D., Savi, M., Pederzolli, F., Siracusa, D., *et al.* (2022). Design and development of network monitoring strategies in p4-enabled programmable switches. *Proceedings of the IEEE/IFIP Network Operations and Management Symposium 2022. Network and Service Management in the Era of Cloudification, Softwarization and Artificial Intelligence*, 1–6 (see p. 25).

Ding, D., Savi, M., Siracusa, D., *et al.* (2020). Estimating logarithmic and exponential functions to track network traffic entropy in P4. *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 1–9 (see p. 14).

DPDK Project. (n.d.). *DPDK Programmer's Guide: Poll Mode Driver*. Retrieved October 12, 2026, from https://doc.dpdk.org/guides/prog_guide/ethdev/ethdev.html (see pp. 10, 14, 15).

Duberstein, T., Decina, A., Tucker, D., R, M., dependabot[bot], ajwerner, Stoycos, A., Guillemard, M., Wu, T., Tuetuopay, Findlay, W., Nakayama, K., Everton, D., Fredette, A., Andres, JEROME, Q., Han, R., Thia, yesho, … Trishchenkov, M. (2025, November). aya-rs/aya. https://github.com/aya-rs/aya (see p. 16).

Evripidou, S., Li, X., Alquran, M., Cheng, R., Taha, A., Imran, M., Flynn, D., & Pezaros, D. (2026). Organisational Cybersecurity Challenges in Digital Twin Development: A Critical Analysis and Research Directions. *Computers & Security*, 105001 (see p. iii).

Feamster, N., Rexford, J., & Zegura, E. (2014). The road to SDN: an intellectual history of programmable networks. *ACM SIGCOMM Computer Communication Review*, *44*(2), 87–98 (see pp. 2, 7).

Feng, Y., Chen, Z., Song, H., Zhang, Y., Zhou, H., Sun, R., Dong, W., Lu, P., Liu, S., Zhang, C., *et al.* (2024). Empower programmable pipeline for advanced stateful packet processing. *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, 491–508 (see p. 14).

Ferguson, A. D., Gribble, S., Hong, C.-Y., Killian, C., Mohsin, W., Muehe, H., Ong, J., Poutievski, L., Singh, A., Vicisano, L., *et al.* (2021). Orion: Google's {Software-Defined} networking control plane. *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 83–98 (see p. 2).

Fink, I. B., Köhler, W., Serror, M., & Wehrle, K. (2025). P4ward: Fine-grained behavioral policy enforcement for industrial networks. *2025 IEEE 50th Conference on Local Computer Networks (LCN)*, 1–10 (see p. 61).

Franco, D., Zaballa, E. O., Zang, M., Atutxa, A., Sasiain, J., Pruski, A., Rojas, E., Higuero, M., & Jacob, E. (2024). A comprehensive latency profiling study of the Tofino P4 programmable ASIC-based hardware. *Computer Communications*, *218*, 14–30 (see p. 22).

Girdhar, M., Park, K., Su, W., Hong, J., & Liu, C.-C. (2025). Cybersecurity Enhancement in Digital Substations: Hidden Markov Model-Based Smart Cyber Switching and Threat Response. *IEEE Access*, *13*, 217022–217037. https://doi.org/10.1109/ACCESS.2025.3648339 (see pp. 56–59).

Haleplidis, E., Pentikousis, K., Denazis, S., Hadi Salim, J., Meyer, D., & Koufopavlou, O. (2015, January). Software-Defined Networking (SDN): Layers and Architecture Terminology. https://doi.org/10.17487/RFC7426 (see p. 8).

Halpern, J., & Pignataro, C. (2015, October). Service Function Chaining (SFC) Architecture. https://doi.org/10.17487/RFC7665 (see p. 7).

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., & Menth, M. (2023). A survey on data plane programming with p4: Fundamentals, advances, and applied research. *Journal of Network and Computer Applications*, *212*, 103561 (see pp. 3, 10, 12–14, 17, 18).

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. *Proceedings of the 14th international conference on emerging networking experiments and technologies*, 54–66 (see pp. 4, 15, 18, 19).

Holik, F. (2024). SGFabric: An emulated Smart Grid Network orchestrated by an eBPF-based BPFabric and a Dashboard application [Accessed: 2026-04-12]. (See p. 69).

Holik, F., Cook, M. M., Li, X., Shah, A. A., & Pezaros, D. (2025). Programmable Data Planes for Increased Digital Resilience in OT Networks. *IEEE Communications Magazine*, *63*(7), 162–169. https://doi.org/10.1109/MCOM.001.2400446 (see pp. iii, 3, 16, 19, 56).

Holik, F., Jouet, S., & Pezaros, D. (2025). An eBPF-Based Programmable Network Architecture for OT Digital Resilience Use-Cases. *2025 IEEE Symposium on Computers and Communications (ISCC)*, 1–6 (see pp. 16, 19, 26, 28, 30).

Holik, F., Mcilwraith, K., Shah, A. A., & Pezaros, D. P. (2025). In-Network GOOSE Encryption with eBPF-based Programmable Network Architecture. *2025 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, 1–6. https://doi.org/10.1109/SmartGridComm65349.2025.11204611 (see p. 58).

Hoyos, J., Dehus, M., & Brown, T. X. (2012). Exploiting the GOOSE protocol: A practical attack on cyber-infrastructure. *2012 IEEE Globecom workshops*, 1508–1513 (see pp. 56–58).

Intel Corporation. (2021, April). *$P_{4_{16}}$ Intel® Tofino™ Native Architecture – Public Version* [Document Number: 631348-0001]. Intel Corporation. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf (see p. 13).

Jin, C., Tang, J., & Ghosh, P. (2013). Optimizing electric vehicle charging with energy storage in the electricity market. *IEEE Transactions on Smart Grid*, *4*(1), 311–320 (see p. 62).

Jin, X., Li, X., Zhang, H., Foster, N., Lee, J., Soulé, R., Kim, C., & Stoica, I. (2018). {NetChain}:{Scale-Free}{Sub-RTT} coordination. *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 35–49 (see pp. 23, 27).

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*, 121–136 (see pp. 23, 27).

Jouet, S., & Pezaros, D. P. (2017). BPFabric: Data plane programmability for software defined networks. *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 38–48 (see pp. 2, 69).

Kaplan, A., & Feibish, S. L. (2022). Practical handling of DNS in the data plane. *Proceedings of the Symposium on SDN Research*, 59–66 (see pp. 14, 43, 44, 50).

Karkashina, T., Shah, A. A., & Pezaros, D. P. (2024). In-network real-time flow classification using hierarchical decision trees. *2024 20th International Conference on Network and Service Management (CNSM)*, 1–5 (see pp. 25, 28).

Kianpisheh, S., & Taleb, T. (2022). A survey on in-network computing: Programmable data plane and technology specific applications. *IEEE Communications Surveys & Tutorials*, *25*(1), 701–761 (see p. 3).

Kicinski, J., & Viljoen, N. (2016). eBPF Hardware Offload to SmartNICs: cls bpf and XDP. *Proceedings of netdev*, *1* (see p. 18).

Kim, D., Liu, Z., Zhu, Y., Kim, C., Lee, J., Sekar, V., & Seshan, S. (2020). TEA: Enabling state-intensive network functions on programmable switches. *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, 90–106 (see pp. 23, 27, 44, 54).

Kim, J., Kim, H., & Rexford, J. (2021). Analyzing traffic by domain name in the data plane. *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, 1–12 (see pp. 43, 44).

Kohler, E., Morris, R., Chen, B., Jannotti, J., & Kaashoek, M. F. (2000). The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, *18*(3), 263–297 (see pp. 10, 14).

Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., & Uhlig, S. (2014). Software-defined networking: A com-

prehensive survey. *Proceedings of the IEEE*, *103*(1), 14–76 (see pp. 2, 3, 9).

Laki, S., Stoyanov, R., Kis, D., Soulé, R., Vörös, P., & Zilberman, N. (2021). P4Pi: P4 on Raspberry Pi for networking education. *ACM SIGCOMM Computer Communication Review*, *51*(3), 17–21 (see p. 18).

Lapolli, Â. C., Marques, J. A., & Gaspary, L. P. (2019). Offloading real-time DDoS attack detection to programmable data planes. *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 19–27 (see p. 57).

Li, J., Nelson, J., Michael, E., Jin, X., & Ports, D. R. (2020). Pegasus: Tolerating skewed workloads in distributed storage with {In-Network} coherence directories. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 387–406 (see pp. 23, 27).

Li, W., Bütün, B., Chu, T., Fiore, M., & Patras, P. (2025). Pallas: A Data-Plane-Only Approach to Accurate Persistent Flow Detection on Programmable Switches in High-Speed Networks. *2025 IEEE 33rd International Conference on Network Protocols (ICNP)*, 1–11 (see pp. 14, 25, 28).

Li, W., Chu, T., Bouganis, C.-S., & Patras, P. (2025). Harmonia: A swift and accurate approximate data structure for real-time heavy flow detection in high-speed networks. *International Conference on Advanced Data Mining and Applications*, 234–243 (see pp. 14, 25, 28).

Li, W., Li, X., Pezaros, D., & Patras, P. (2026). Charon: Stratified Priority Sampling for Differentiated Per-Flow Measurement in High-Speed Networks. *Proceedings of the ACM SIGCOMM 2026 Conference: Posters and Demos* (see pp. iii, 25, 28).

Li, X., & Pezaros, D. (2025). Dataplane-Only DNS Caching in P4. *2025 IEEE Symposium on Computers and Communications (ISCC)*, 1–6. https://doi.org/10.1109/ISCC65549.2025.11326441 (see pp. iii, 31).

Liu, C., Tak, B., & Wang, L. (2024). Understanding performance of ebpf maps. *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, 9–15 (see p. 66).

Liu, Z., Bai, Z., Liu, Z., Li, X., Kim, C., Braverman, V., Jin, X., & Stoica, I. (2019). {DistCache}: Provable load balancing for {Large-Scale} storage systems with distributed caching. *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 143–157 (see pp. 23, 27).

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. *2007 IEEE International Conference on Microelectronic Systems Education (MSE'07)*, 160–161 (see p. 2).

Mahboob, T., Holik, F., Shah, A. A., & Pezaros, D. (2025). Adaptive Learning Feature Quantization for In-network FDI Detection in IEC 61850 Digital Substations. *2025 IEEE International Conference on Communications, Control, and Computing Technologies for Smart Grids (SmartGridComm)*, 1–6 (see p. 58).

Mahboob, T., Li, M., Shah, A. A., & Pezaros, D. (2026). Enhancing Smart Grid Cyber Resilience Against FDI Attacks Using Multi-Agent Recurrent DDPG. *Network* (see p. 68).

McCanne, S., & Jacobson, V. (1993). The BSD Packet Filter: A New Architecture for User-level Packet Capture. *USENIX winter*, *46*, 259–270 (see p. 15).

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., & Turner, J. (2008). OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM computer communication review*, *38*(2), 69–74 (see pp. 2, 9).

Miano, S., Sanaee, A., Risso, F., Rétvári, G., & Antichi, G. (2022). Domain specific run time optimization for software data planes. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 1148–1164. https://doi.org/10.1145/3503222.3507769 (see pp. 26, 28).

Michel, O., Bifulco, R., Retvari, G., & Schmid, S. (2021). The programmable data plane: Abstractions, architectures, algorithms, and applications. *ACM Computing Surveys (CSUR)*, *54*(4), 1–36 (see pp. 1–3, 10, 17).

Minerva, R., Lee, G. M., & Crespi, N. (2020). Digital Twin in the IoT Context: A Survey on Technical Features, Scenarios, and Architectural Models. *Proceedings of the IEEE*, *108*(10), 1785–1824. https://doi.org/10.1109/JPROC.2020.2998530 (see p. 58).

Mockapetris, P. (1987a, November). Domain names - concepts and facilities. https://doi.org/10.17487/RFC1034 (see p. 32).

Mockapetris, P. (1987b, November). Domain names - implementation and specification. https://doi.org/10.17487/RFC1035 (see pp. 31, 32).

MZ Automation GmbH. (2024). libiec61850: Open-source IEC 61850 library [Version 1.6.1, accessed April 2026]. https://github.com/mz-automation/libiec61850 (see p. 62).

Namkung, H., Liu, Z., Kim, D., Sekar, V., & Steenkiste, P. (2022). SketchLib: Enabling Efficient Sketch-based Monitoring on Programmable Switches. *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 743–759. https://www.usenix.org/conference/nsdi22/presentation/namkung (see pp. 14, 25, 28).

Nasr, T., Torabi, S., Bou-Harb, E., Fachkha, C., & Assi, C. (2023). ChargePrint: A Framework for Internet-Scale Discovery and Security Analysis of EV Charging Management Systems. *Proceedings of the Network and Distributed System Security Symposium (NDSS)* (see p. 62).

P4 Language Consortium. (2016, January). *$P4_{14}$ Language Specification, Version 1.1.0* (tech. rep.) (Available at https://p4.org/specifications/). P4 Language Consortium. Retrieved April 12, 2026, from https://p4.org/wp-content/uploads/sites/53/p4-spec/p4-14/v1.1.0/tex/p4.pdf (see p. 10).

P4 Language Consortium. (2022, December). *$P4_{16}$ Portable Switch Architecture (PSA), Version 1.2* (tech. rep.) (Available at https://p4.org/specifications/). P4 Language Consortium. Retrieved April 12, 2026, from https://p4.org/wp-content/uploads/sites/53/p4-spec/docs/PSA-v1.2.pdf (see p. 13).

P4 Language Consortium. (2024, October). *$P4_{16}$ Language Specification, Version 1.2.5* (tech. rep.) (Available at https://p4.org/specifications/). P4 Language Consortium. Retrieved April 12, 2026, from https://p4.org/wp-content/uploads/sites/53/2024/10/P4-16-spec-v1.2.5.pdf (see pp. 11, 42, 43).

Pittalà, G. F., Rinieri, L., Al Sadi, A., Davoli, G., Melis, A., Prandini, M., & Cerroni, W. (2024). Leveraging Data Plane Programmability to enhance service orchestration at the edge: A focus on industrial security. *Computer Networks*, *246*, 110397. https://doi.org/https://doi.org/10.1016/j.comnet.2024.110397 (see p. 61).

Pochat, V. L., Van Goethem, T., Tajalizadehkhoob, S., Joosen, W., *et al.* (2018). Tranco: A research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156* (see p. 32).

Ports, D. R., & Nelson, J. (2019). When should the network be the computer? *Proceedings of the Workshop on Hot Topics in Operating Systems*, 209–215 (see p. 3).

Puttlitz, C., Parizotto, R., & Schaeffer-Filho, A. (2024). P4NetIntel: End-to-End Network Telemetry with eBPF and XDP. *2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 1–6. https://doi.org/10.1109/NFV-SDN61811.2024.10807468 (see pp. 24, 28).

Qian, M., Cui, L., Tso, F. P., Deng, Y., Zhang, Z., & Jia, W. (2025). FlxVRM: Enabling Online Configuring Memory Via Virtualization on Programmable Data Plane. *IEEE Transactions on Services Computing* (see p. 14).

Rivera, S., Goetz, S. M., Kouro, S., Lehn, P. W., Pathmanathan, M., Bauer, P., & Mastromauro, R. A. (2022). Charging infrastructure and grid integration for electromobility. *Proceedings of the IEEE*, *111*(4), 371–396 (see p. 62).

Rizzo, L. (2012). netmap: A Novel Framework for Fast Packet I/O. *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, 101–112. https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo (see pp. 10, 14, 15).

Sagkriotis, S., & Pezaros, D. (2022). Scale-friendly in-network coordination. *GLOBECOM 2022-2022 IEEE Global Communications Conference*, 5747–5752 (see pp. 23, 27).

Şahin, M. E., & Demirci, M. (2025). HELP4DNS: Leveraging the programmable data plane for effective and robust defense against DDoS attacks on DNS. *Journal of Network and Computer Applications*, *240*, 104198 (see pp. 24, 27).

Saltzer, J. H., Reed, D. P., & Clark, D. D. (1984). End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, *2*(4), 277–288. https://web.mit.edu/saltzer/www/publications/endtoend/endtoend.pdf (see p. 79).

Sapio, A., Canini, M., Ho, C.-Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D., & Richtarik, P. (2021). Scaling Distributed Machine Learning with In-Network Aggregation. *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 785–808. https://www.usenix.org/conference/nsdi21/presentation/sapio (see p. 79).

Scazzariello, M., Caiazzi, T., Ghasemirahni, H., Barbette, T., Kostić, D., & Chiesa, M. (2023). A {High-Speed} stateful packet processing approach for tbps programmable switches. *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 1237–1255 (see pp. 14, 22, 54, 56).

Scott-Hayward, S., Natarajan, S., & Sezer, S. (2015). A survey of security in software defined networks. *IEEE Communications Surveys & Tutorials*, *18*(1), 623–654 (see p. 2).

Shirmarz, A., Bragatto, M. N., Verdi, F. L., Singh, S. K., Rothenberg, C. E., Patra, G., & Pongrácz, G. (2025). In-Network AR/CG Traffic Classification Entirely Deployed in the Programmable Data Plane: Unlocking RTP Features and L4S Integration. *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*, 477–485 (see p. 14).

Siddique, H., Neves, M., Kuzniar, C., & Haque, I. (2021). Towards network-accelerated ML-based distributed computer vision systems. *2021 IEEE 27th International Conference on Parallel and Distributed Systems (ICPADS)*, 122–129 (see pp. 25, 28).

Simon, M., Stubbe, H., Gallenmüller, S., & Carle, G. (2024). Honey for the Ice Bear - Dynamic eBPF in P4. *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions*, 44–50. https://doi.org/10.1145/3672197.3673436 (see pp. 25, 28).

Singh, A., Ong, J., Agarwal, A., Anderson, G., Armistead, A., Bannon, R., Boving, S., Desai, G., Felderman, B., Germano, P., *et al.* (2015). Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. *ACM SIGCOMM computer communication review*, *45*(4), 183–197 (see p. 2).

Sivaraman, A., Mason, T., Panda, A., Netravali, R., & Kondaveeti, S. A. (2020). Network architecture in the age of programmability. *ACM SIGCOMM Computer Communication Review*, *50*(1), 38–44 (see p. 2).

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. *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 571–592 (see pp. 26, 29).

Teng, L., Hung, C.-H., & Wen, C. H.-P. (2022). P4SF: A high-performance stateful firewall on commodity P4-programmable switch. *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 1–5 (see pp. 21, 56).

Terrace, J., & Freedman, M. J. (2009). Object storage on CRAQ: High-throughput chain replication for read-mostly workloads. *USENIX Annual Technical Conference* (see p. 23).

Van Tu, N., Yoo, J.-H., & Hong, J. W.-K. (2019). Building hybrid virtual network functions with express data path. *2019 15th International Conference on Network and Service Management (CNSM)*, 1–9 (see pp. 4, 66).

Vieira, M. A., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., & Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, *53*(1), 1–36 (see pp. 1–3, 16–18, 57).

Williams, T. J. (1994). The Purdue enterprise reference architecture. *Computers in industry*, *24*(2-3), 141–158 (see p. 58).

Wirtgen, T., Rousseaux, T., De Coninck, Q., Rybowski, N., Bush, R., Vanbever, L., Legay, A., & Bonaventure, O. (2023). {xBGP}: Faster innovation in routing protocols. *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, 575–592 (see p. 2).

Woodruff, J., Ramanujam, M., & Zilberman, N. (2019). P4dns: In-network dns. *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 1–6 (see pp. 21, 24, 27, 33, 45–47).

Wragg, D. (2020, September 9). *Unimog – Cloudflare's Edge Load Balancer*. Cloudflare. Retrieved October 12, 2026, from https://blog.cloudflare.com/unimog-cloudflares-edge-load-balancer/ (see p. 15).

Xiong, Z., & Zilberman, N. (2019). Do switches dream of machine learning? toward in-network classification. *Proceedings of the 18th ACM workshop on hot topics in networks*, 25–33 (see pp. 25, 28).

Yegorov, P. K., Lackovitch, A., Dean, E., Mustafa, H. M., Basumallik, S., & Srivastava, A. (2023). Analyzing goose security in iec61850-based substation using ml, sdn and digital twin. *2023 North American Power Symposium (NAPS)*, 1–6 (see p. 58).

yonghong-song, Blanco, B., Gregg, B., 4ast, Qin, T., Goldshtein, S., Marchevsky, D., Chen, H., Chaignon, P., Tao, R., Marchand, J., Nakryiko, A., rockyxing, chantra, Zhang, W., ekyooo, Vásquez, M., Lin, G. C.-P., Das, S., … Babrou, I. (2025, November). iovisor/bcc. https://github.com/iovisor/bcc (see p. 16).

Yoo, S., & Chen, X. (2021). Secure keyed hashing on programmable switches. *Proceedings of the ACM SIGCOMM 2021 Workshop on Secure Programmable network INfrastructure*, 16–22 (see p. 14).

Yoo, S., Chen, X., & Rexford, J. (2024). {SmartCookie}: Blocking {Large-Scale}{SYN} Floods with a {Split-Proxy} Defense on Programmable

Data Planes. *33rd USENIX Security Symposium (USENIX Security 24)*, 217–234 (see pp. 26, 27).

Yuan, Y., Alama, O., Fei, J., Nelson, J., Ports, D. R., Sapio, A., Canini, M., & Kim, N. S. (2022). Unlocking the power of inline {Floating-Point} operations on programmable switches. *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 683–700 (see p. 14).

Zemanek, S., Hacker, I., Wolsing, K., Wagner, E., Henze, M., & Serror, M. (2022). PowerDuck: a GOOSE data set of cyberattacks in substations. *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, 49–53 (see pp. 63, 70).

Zhao, Y., Liu, W., Dong, F., Yang, T., Li, Y., Yang, K., Liu, Z., Jia, Z., & Yang, Y. (2023). P4lru: Towards an lru cache entirely in programmable data plane. *Proceedings of the ACM SIGCOMM 2023 Conference*, 967–980 (see pp. 24, 27, 45, 46).

Zheng, C., Hong, X., Ding, D., Vargaftik, S., Ben-Itzhak, Y., & Zilberman, N. (2023). In-network machine learning using programmable network devices: A survey. *IEEE Communications Surveys & Tutorials*, *26*(2), 1171–1200 (see p. 25).

Zheng, C., Xiong, Z., Bui, T. T., Kaupmees, S., Bensoussane, R., Bernabeu, A., Vargaftik, S., Ben-Itzhak, Y., & Zilberman, N. (2024). IIsy: Hybrid in-network classification using programmable switches. *IEEE/ACM Transactions on Networking*, *32*(3), 2555–2570 (see pp. 25, 28).

Zhu, H., Wang, T., Hong, Y., Ports, D. R., Sivaraman, A., & Jin, X. (2022). {NetVRM}: Virtual register memory for programmable networks. *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 155–170 (see p. 14).

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 (see p. 2).

Zilberman, N., Watts, P. M., Rotsos, C., & Moore, A. W. (2015). Reconfigurable network systems and software-defined networking. *Proceedings of the IEEE*, *103*(7), 1102–1124 (see p. 2).