User-Plane Algorithms for Stateless and Stateful Inference in Programmable Networks

by

Aristide Tanyi-Jong Akem

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in

Telematic Engineering

Universidad Carlos III de Madrid

Tutor/Advisor: Marco Fiore

September 2024

User-Plane Algorithms for Stateless and Stateful Inference in Programmable Networks

Prepared by:
Aristide Tanyi-Jong Akem
IMDEA Networks Institute, Universidad Carlos III de Madrid
contact: aristide.akem@imdea.org

Under the advice of:
Marco Fiore, IMDEA Networks Institute

This work has been supported by:

To my family.

"Call it a clan, call it a network, call it a tribe, call it a family. Whatever you call it, whoever you are, you need one."
– Jane Howard, 1978

Acknowledgements

I thank God for his abundant grace throughout this PhD journey. It has been a challenging but exciting journey and many people have contributed to making it possible.

I am very grateful to my supervisor, Dr. Marco Fiore, who hired me and allowed me to carry out my PhD within his research group. Thanks to his encouragement, guidance and sleepless nights when we had deadlines, we have co-authored several publications, and I have gained a strong research foundation, delivered several talks, peer-reviewed multiple papers, and travelled around the world to attend conferences and visit other research labs.

I also thank Dr. Michele Gucciardo for his guidance and collaboration throughout my PhD studies, which greatly contributed to my success. Without his availability and assistance from the early stages of my PhD, things would have been a lot more difficult. Also, I thank Beyza Bütün for her dedication and collaboration which made things a lot easier for me in the projects we worked on during the second half of my PhD.

Many thanks to the members of the Networks Data Science Group at IMDEA for the group meetings, lunches and ping-pong games that kept life at work interesting. I thank the HR and Admin at IMDEA for their support in navigating the numerous bureaucratic procedures that I went through. I also thank everyone at IMDEA for contributing to the conducive research environment and for their regular feedback on my work.

I would also like to thank Dr. Guillaume Fraysse for supervising my stay at Orange Labs in Paris and for ensuring that we co-authored a paper. Many thanks to Joanna Balcerzak and Dr. Bertrand Decocq who helped in arranging the visit and to everyone in the SMART team for welcoming me. I thank Dr. Ian Wassell and Dr. Kan Lin who arranged and supervised my stays at The University of Cambridge and Ranplan Wireless respectively. I also thank the ESRs of the BANYAN project and our project managers.

I thank my religious communities at OLM in Madrid, Santa Beatriz in Leganés, Groupe Afrique in Créteil, and Fisher House in Cambridge. I also thank the Taku family in Cambridge and Hillary's family in Madrid for their never-ending hospitality.

A big thank you to all my friends and loved ones, especially those who kept in touch and genuinely cared about me even when I got too busy. Finally and very importantly, I am very grateful to my family to whom I have dedicated this thesis. You have always believed in me and are a source of inspiration when things get tough. Thank you!

Published and Submitted Content

This thesis is based on the following published or submitted papers:

[1] Aristide Tanyi-Jong Akem and Marco Fiore, "Towards Data-Driven Management of Mobile Networks through User Plane Inference". In: *NOMS 2024 - IEEE/IFIP Network Operations and Management Symposium*, Seoul, South Korea, 6-10 May 2024, pp. 1-4. https://doi.org/10.1109/NOMS59830.2024.10575655.

- This work is fully included and the contents are reported in Chapters 1 and 3.
- The thesis author prepared and wrote the entire manuscript.
- The material from this source included in this thesis is not singled out with typographic means and references.

[2] Aristide Tanyi-Jong Akem, Michele Gucciardo and Marco Fiore, "Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests". In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, NY, USA, 2023, pp. 1–10, doi: https://doi.org/10.1109/INFOCOM53939.2023.10229100.

- This work is fully included and the contents are reported in Chapters 2 and 6.
- The thesis author is the first author of this work and led the design of the Flowrest framework, its implementation and experimentation in hardware, and participated in writing several parts of the manuscript.
- The material from this source included in this thesis is not singled out with typographic means and references.

[3] Aristide Tanyi-Jong Akem, Michele Gucciardo and Marco Fiore, "Ultra-Low Latency User-Plane Cyberattack Detection in SDN-based Smart Grids". In: *The 15th ACM International Conference on Future and Sustainable Energy Systems (E-Energy '24)*, June 04–07, 2024, Singapore, Singapore. ACM, New York, NY, USA 7 Pages. https://doi.org/10.1145/3632775.3661995

- This work is fully included and the contents are reported in Chapter 4.
- The thesis author is the first author of this work and led the design and experimentation of the solution, as well as writing the manuscript.

- The material from this source included in this thesis is not singled out with typographic means and references.

[4] Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo, and Marco Fiore, "Henna: Hierarchical Machine Learning Inference in Programmable Switches". In: *The Proceedings of the 1st International Workshop on Native Network Intelligence (NativeNi'22)*, Roma, Italy, 2022, doi: https://doi.org/10.1145/3565009.3569520.

- This work is fully included and the contents are reported in Chapter 5.
- The thesis author led the conceptual design of the Henna framework, led the implementation of the first stage of the solution, took part in running experiments, generating plots, and writing several parts of the manuscript.
- The material from this source included in this thesis is not singled out with typographic means and references.

[5] Aristide Tanyi-Jong Akem, Guillaume Fraysse and Marco Fiore, "Encrypted Traffic Classification at Line Rate in Programmable Switches with Machine Learning". In: *NOMS 2024 - IEEE/IFIP Network Operations and Management Symposium*, Seoul, South Korea, 6-10 May 2024. https://doi.org/10.1109/NOMS59830.2024.10575394.

- This work is fully included and the contents are reported in Chapters 6 and 7.
- The thesis author is the first author of this work and led the design of the encrypted traffic classification framework, its implementation and experimentation in hardware, and participated in writing most parts of the manuscript.
- The material from this source included in this thesis is not singled out with typographic means and references.

[6] Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo and Marco Fiore, "Practical and General-Purpose Flow-Level Inference with Random Forests in Programmable Switches". In: *IEEE/ACM Transactions on Networking*, 2024. (Submitted)

- This work is partly included and the contents are reported in Chapters 6 and 7.
- The thesis author is the first author of this work and led the design of the Flowrest framework, its implementation and experimentation in hardware, the reproduction of benchmarks, the demonstration of Flowrest's ability to accommodate any tree mapping, and took part in writing parts of the paper.
- The material from this source included in this thesis is not singled out with typographic means and references.

[7] **Aristide Tanyi-Jong Akem**, Beyza Bütün, Michele Gucciardo and Marco Fiore, "Showcasing In-Switch Machine Learning Inference". In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, Madrid, Spain, 2023, pp. 299-301, doi: https://doi.org/10.1109/NetSoft57336.2023.10175464. **(🏆 Best Demo)**

- This demo is fully included and the contents are reported in Chapter 7.
- The thesis led the design of the Flowrest framework which is demonstrated in this work, took part in the implementation and experimentation and participated in writing several parts of the manuscript.
- The material from this source included in this thesis is not singled out with typographic means and references.

[8] **Aristide Tanyi-Jong Akem***, Beyza Bütün*, Michele Gucciardo and Marco Fiore, "Jewel: Resource-Efficient Joint Packet and Flow Level Inference in Programmable Switches". In: *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, Vancouver, Canada, 2024, pp. 1-10. (Accepted) https://hdl.handle.net/20.500.12761/1777. ** Equal contributors*

- This work is partly included and the contents are reported in Chapter 8.
- The thesis author is a co-first author of this work and led the conceptual design of the joint flow and packet level model in Python, led the reproduction and experimentation of the benchmark solutions, took part in generating plots, and wrote several parts of the manuscript.
- The material from this source included in this thesis is not singled out with typographic means and references.

Other publications that are not a major part of this thesis include:

[9] Michele Gucciardo, Beyza Bütün, **Aristide Tanyi-Jong Akem**, and Marco Fiore, "Evaluating the Impact of Flow Length on the Performance of In-Switch Inference Solutions". In: *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, Vancouver, Canada, 2024, pp. 1-6. (Accepted) https://hdl.handle.net/20.500.12761/1800.

[10] Beyza Bütün, **Aristide Tanyi-Jong Akem**, Michele Gucciardo and Marco Fiore, "Fast Detection of Cyberattacks on the Metaverse through User-plane Inference". In: *2023 IEEE International Conference on Metaverse Computing, Networking and Applications (MetaCom)*, Kyoto, Japan, 2023, pp. 350-354, doi: https://doi.org/10.1109/MetaCom57706.2023.00067.

[**11**] Michele Gucciardo, **Aristide Tanyi-Jong Akem**, Beyza Bütün and Marco Fiore, "Demonstrating Flow-Level In-Switch Inference". In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, NJ, USA, 2023, pp. 1-2, doi: https://doi.org/10.1109/INFOCOMWKSHPS57453.2023.10225967.

Abstract

Over the past decade, network complexity has grown exponentially to support the emergence of new and innovative applications. This increased sophistication has rendered most human-in-the-loop approaches to network management tasks obsolete, calling for more automation and flexibility in the network. The advent of Software-Defined Networking (SDN) with a programmable control plane was a huge step in the right direction, giving rise to a variety of network automation applications running in the SDN control plane. With the vulgarization of Machine Learning (ML) in recent years, many network automation applications use ML techniques to address network problems like intrusion detection, routing optimization, quality of service prioritization, and fault detection. Yet, as these applications run in the control plane, many of them cannot respond in real-time to network issues and incur a response delay in the order of milliseconds to seconds, which is undesirable in ultra-low-latency applications that will abound in 6G networks.

Recent advances in user-plane programmability have led to the current availability of off-the-shelf programmable user-plane equipment like Intel Tofino switches, alongside compatible network programming languages like P4. This has sparked a strong interest in in-network computation, with efforts to offload ML models from the control plane to the user plane to reduce their response time and enable inference at line rate with low latency and high throughput. Most work on user-plane inference has focused on programmable switches due to their ubiquitous presence in the network and the availability of multiple high-speed ports. However, switches are highly constrained in terms of available memory, support for mathematical operations, and the number of allowed operations per packet. This makes it impossible to train ML models in the switch and shifts the focus to deploying trained models into user-plane switches. The constraints above also make complex models like Neural Networks (NN) less feasible for in-switch deployment. Instead, most prior works have deployed tree-based models like Decision Trees (DT) and Random Forests (RF) for in-switch inference due to their simple logical structure and few operations required at inference, which make them ideal for constrained environments. Yet, these works have several limitations such as limited scalability and adaptability which translate into performance barriers when handling complex inference tasks.

This thesis proposes efficient solutions for embedding ML models into production-grade programmable switches, thereby addressing the above limitations and advancing the state of the art in ML-based user-plane inference.

To illustrate the evolution across the solutions presented in the thesis, a practical application of user-plane inference is considered to show how in-switch inference can enable rapid detection of cyberattacks on SDN-based Smart Grid (SG) networks. Current power grids are smart, with millions of electronic devices interconnected by data networks. This exposes them to many cyberattacks which could lead to power outages and data breaches with far-reaching impacts. Thus, the timely detection of cyberattacks is critical. ML models are widely used for cyberattack detection in SDN-based SGs, where the models either run in external servers or in-network but fully in the control plane or distributed between the control and user planes. In these cases, the models do not run at line rate and incur millisecond-level delays in attack detection. The application developed in this thesis explores how ML inference in programmable switches at Packet-Level (PL) can enable accelerated attack detection and mitigation in SGs at line rate with sub-microsecond delay. The proposed workflow brings the concept of user-plane inference to SDN-based SGs for the first time, and deploys a trained DT model into the switch pipeline for real-time inference on live traffic. Results produced in this thesis show how a pure user-plane solution achieves up to 99% accuracy in attack detection and classification, while operating up to four orders of magnitude faster than solutions running entirely in the control plane.

The above solution and all earlier solutions for PL inference in the user-plane focus on flat classification, and have significant structural limitations that prevent them from scaling when handling complex inference tasks. To tackle these limitations, this thesis proposes `Henna`, the pioneer implementation of an in-switch multi-stage hierarchical classification system. The concept upon which `Henna` hinges is that of splitting a difficult classification task into easier cascaded inference tasks, which can then be addressed with separate resource-efficient tree-based classifiers. The design of `Henna` aligns with the internal organization of the Protocol Independent Switch Architecture (PISA), and integrates state-of-the-art strategies for mapping decision trees to switch hardware. `Henna` is then implemented into a real-world testbed with off-the-shelf Intel Tofino programmable switches using the P4 language. Experiments with a complex 21-category classification task based on measurement data exhibit how `Henna` improves the F1-score of an advanced single-stage model by 21%, while maintaining usage of switch resources at 8% on average.

Despite the improvements brought about by `Henna`, existing hardware-compatible in-switch inference solutions are still either limited to only PL operation, lack support for rich statistical features, or are not scalable, hitting performance barriers in complex tasks involving large decision spaces. To address this limitation, `Flowrest` is presented as a first complete RF model implementation that operates at the level of individual flows in

commercial switches. The proposed solution builds on (*i*) novel guidelines for tailoring RF models to operation in programmable switches right from the design phase, (*ii*) an original framework to embed flow-level (FL) machine learning models into programmable switch ASICs, and (*iii*) efficient strategies for maintaining state within switches to compute, store and employ FL features for inference. `Flowrest` is implemented in a hardware switch as an open-source software using the P4 language.

`Flowrest` sets a new standard for FL inference in the user plane. To validate this claim, a thorough evaluation of the proposed solution is conducted in an experimental platform based on Intel Tofino switches in two steps; (*i*) `Flowrest` is evaluated on unencrypted traffic, comparing it to major existing proposals for in-switch inference which all target unencrypted traffic, and (*ii*) it is then evaluated on encrypted traffic classification. Results from the evaluation with tasks of unprecedented complexity show how `Flowrest` achieves accuracy gains in the $10\% - 39\%$ range over previous approaches to implement DT and RF models in real-world equipment.

Despite the improved performance resulting from FL classification, a major dichotomy still exists between works for in-switch inference, based on whether they operate at PL or FL. The former relies on simple features from packet headers that are simple to implement but limit accuracy in challenging use cases; the latter exploits richer flow-based statistical features to improve accuracy, but leaves early packets in each flow unclassified. To close this gap, this thesis presents `Jewel`, an in-switch ML solution based on a fully joint PL and FL design, which offers the best of both worlds by classifying early flow packets individually at PL and shifting to FL inference as soon as possible. The proposed solution involves (*i*) a single RF model trained to classify both packets and flows, and (*ii*) hardware-aware model selection and training techniques for resource footprint minimization. `Jewel` is implemented in P4 and deployed in a testbed with Intel Tofino switches, where extensive experiments are conducted with a variety of real-world use cases. Results from experiments conducted in this thesis reveal how `Jewel` outperforms four state-of-the-art benchmarks, with absolute accuracy gains in the $2.0\% - 5.3\%$ range, while consuming a modest amount of switch resources.

In summary, this thesis proposes novel solutions for inference in programmable network user planes. Technical details on the design and implementation of the proposed solutions are described first, followed by thorough experimental evaluations that shed light on the merits of each solution in comparison to prior work. Through these contributions, this thesis sets new standards in user-plane ML inference and makes steps towards enabling and encouraging the pervasive adoption of user-plane inference in programmable networks by making all the solutions open-source.

Table of Contents

List of Tables

List of Figures

List of Acronyms

AI Artificial Intelligence

API Application Programming Interface

ASIC Application-Specific Integrated Circuit

BMv2 Behavioural Model version 2

BNN Binarized Neural Networks

BRI Barefoot Runtime Interface

CRC Cyclic Redundancy Check

DL Deep Learning

DPI Deep Packet Inspection

DT Decision Tree

DNP3 Distributed Network Protocol 3

ETC Encrypted Traffic Classification

FL Flow Level

FNR False Negative Rate

FPR False Positive Rate

IAT Inter-Arrival Time

IMA Instant Messaging Application

IoT Internet-of-Things

KNN K-Nearest Neighbours

LR Logistic Regression

M/A Match & Action

MAT Match & Action Table

MAU Match & Action Unit

ML Machine Learning

NFV Network Function Virtualization

NN Neural Networks

PHV Packet Header Vector

PISA Protocol Independent Switch Architecture

PL Packet Level

RA Register Action

RF Random Forest

SCADA Supervisory Control and Data Acquisition

SDE Software Development Emulator

SDN Software Defined Networking

SG Smart Grid

SmartNIC Smart Network Interface Card

SRAM Static Random-Access Memory

TCAM Ternary Content-Addressable Memory

TNA Tofino Native Architecture

TNR True Negative Rate

TPR True Positive Rate

VPN Virtual Private Network

1 Introduction

In recent years, growing network complexity and flexibility have rendered human-in-the-loop approaches to network management impractical, necessitating increased automation to reduce or completely eliminate human intervention. Concepts like self-driving networking [12, 13] or zero-touch network and service management (ZSM) [14, 15] come to the rescue, by employing controllers and orchestrators to collect live traffic measurements from networks, and analyze them to design policies and algorithms that can take effective management decisions in real time or in anticipation.

Artificial Intelligence (AI) techniques like Machine Learning (ML) have become key players in the journey towards increased automation in the planning, deployment and management of modern high-speed networks [16]. In the context of Software Defined Networking (SDN), ML models for network automation are traditionally run in the programmable control plane or in offline ML servers, with a wide range of proposals, *e.g.*, for Quality of Service (QoS) management, routing optimization, and anomaly detection, as recently surveyed [17, 18]. While these models make significant strides towards enhancing the automation of network operations, control plane models suffer from inherent limitations that prevent them from running at line rate, which is a key requirement for self-driving networking [13]. Notably, these models often require communication with the user plane to run inference in real-time on live network traffic, thereby incurring millisecond-level delays [19]. Thus, these models do not meet the strict latency requirements of modern 5G/6G networks [20] or complex low-latency applications, *e.g.*, augmented and virtual reality (AR/VR) [21], and autonomous driving [22].

Over the last decade, SDN user planes have become programmable with available commercial off-the-shelf products like the Intel Tofino Application-Specific Integrated Circuit (ASIC) [23] and the NVIDIA BlueField Data Processing Unit (DPU) [24], alongside domain-specific network programming languages such as P4 [25]. This has given rise to a flurry of applications that exploit user-plane programmability to deploy intelligent in-network functions, *e.g.*, for in-band network telemetry, failure recovery, traffic management, and advanced routing, as reviewed in recent surveys [26, 27].

1.1. Motivation

User-plane programmability has also sparked a strong interest in fully offloading ML models to the user plane to achieve line-rate inference through in-network computation [28], which can potentially reduce inference delay to sub-microsecond levels. This is enabled by the fact that deploying inference models into the user plane allows for inference on packets at the same rate at which they are forwarded through the network, maximizing inference speed. Yet, porting models to user planes is a daunting task owing to strict inherent constraints of user planes such as low available memory, limited support for mathematical operations, and limited number of allowed operations per packet [29].

The constraints above rule out the possibility of training ML models fully in the user plane with P4, and limit user-plane ML applications to the deployment of pre-trained ML models for line rate inference as recently surveyed [30, 31]. The constraints have also forced some prior work to opt for a partial offload, where only feature extraction and/or preliminary traffic filtering occurs in the user plane, while the inference process is kept in the controller [32–35]. However, these approaches still do not run at line rate and incur millisecond-level delays during the back-and-forth inter-plane communication.

Most of the work on full in-network inference with ML has focused on deploying models into programmable switches, due to their ubiquitous presence in the network. One of the earliest works is N2Net [36], which proposed the in-switch deployment of Binarized Neural Networks (BNN), which are Neural Networks (NN) with $+1/-1$ weights and sign activations [37]. While BNNs are simpler than full NNs, deploying them into commodity switches proves to be impractical, as a basic BNN with two layers of 64 and 32 neurons completely exhausts the resources of a Tofino ASIC [36]. Other works also explored the deployment of other models, *e.g.*, Support Vector Machines (SVM), XGBoost, Naive Bayes, or K-Means, but they did not offer good performance and scalability [38–40].

Due to their basic logical structure which is somewhat analogous to the switch pipeline, and the simplicity of the operations involved at inference, Decision Tree (DT) and Random Forest (RF) models have proven to be the most amenable to in-switch operation with a number of proposals to that effect [38, 41–44]. However, most of these works suffer several limitations in terms of design, practicality, and generalizability, which translate into performance and scalability barriers when handling complex classification tasks.

This thesis leverages ML and user-plane programming with P4 to advance the state of the art in user-plane inference by proposing novel and innovative solutions for efficiently designing and embedding DT and RF models into programmable switches for line-rate inference in programmable networks, specifically in the user plane.

1.2. Research challenges

As briefly discussed in Section 1.1, most existing proposals for deploying ML models in programmable network user planes focus on programmable switch targets which are highly constrained in terms of memory, support of mathematical operations, and number of allowed operations per packet. In addition, the P4 language specification [45] presents several constraints of the language that further reduce the sample space of operations and hence the models that can be deployed in the switch using P4. These constraints constitute major challenges to user-plane inference and are described next.

Low available memory. To support high-speed packet processing, *e.g.*, at 6 Tbps with Intel Tofino first generation switch ASICs, expensive memory like Static Random-Access Memory (SRAM) and Ternary Content-Addressable Memory (TCAM) are required. These memories are only available in limited amounts, typically a few hundred megabytes of SRAM and a few megabytes of TCAM. To deploy user-plane inference models in switches, the available memory must be shared between normal switch forwarding functions and the deployed models. This poses a challenge that must be accounted for when considering user-plane inference deployments.

Limited support for mathematical operations. Only basic operations are allowed in user-plane hardware. Typically, additions, bit shifts, and logical operations are supported, whereas division and floating-point operations in general are not allowed. As such, deploying any ML in the user plane will require converting any unsupported operations into simpler ones, *e.g.*, converting a multiplication into multiple additions, or converting a division by a multiple of 2 into a binary shift. This also renders deployment of NN-based models difficult, since they require several mathematical operations, and motivates the choice of simpler DT and RF for in-switch deployment.

Limited number of allowed operations per packet. To ensure line-rate packet processing, the switch must process and forward packets within only tens to hundreds of nanoseconds. Thus, most operations can only be performed once per packet and constructs like loops which do not have a well-known upper bound on the number of allowed operations are not supported [29]. While some hardware targets support packet recirculation by enabling a packet to go through the switch pipeline multiple times, this increases processing cost, and recirculating all packets would cut throughput in half. User-plane models must therefore run operations only once per packet for best results.

In addition to the above constraints, each specific user-plane target also has particular constraints that have to be accounted for when writing P4 programs for them, *e.g.*, the maximum number of bits that can fit in a unit of switch memory or the size and structure of header fields that can be defined. Also, as the P4 language is designed for user-plane targets, it inherently includes some restrictions to accommodate the constraints above. Specifically, P4 programs do not support loops except in the parser and do not

enable inspection of packet payloads. These language-specific constraints must also be accommodated when designing solutions for in-switch inference.

In the face of the above challenges, a key question arises, *i.e.*, *how can the constraints of programmable user-plane equipment be accommodated to efficiently deploy ML models for high-speed inference at line rate, with high throughput and low latency?*

The main objective of this PhD thesis is to answer the above question and contribute to the global vision of self-driving networks [12], by designing efficient solutions for embedding data-driven models into the user plane for high-speed inference. Thus, the above question is split into three sub-questions which are answered in this thesis as follows.

- *What ML models are most suitable for user-plane inference?*
  With the challenges above in mind, a major goal of this thesis is to keep models designed for user-plane inference as simple as possible to ensure their compatibility with user-plane targets. As such, DT and RF models are adopted. In contrast to complex NNs which require multiple operations, tree-based models only require simple comparisons between feature values and model thresholds at tree nodes to arrive at an inference decision. In addition, as it will be demonstrated in Chapter 2, DT and RF models are either on par with or outperform NNs in a variety of inference tasks based on tabular data, as also experimented by existing literature [46]. Thus, they yield the best possible accuracy-complexity trade-off, further motivating their choice for user-plane inference in this thesis.

- *What user plane components should be targeted for model deployments?*
  A variety of off-the-shelf programmable user-plane targets are available on the market including ASICs, *e.g.*, the Intel Tofino [23] ASIC, and the Smart Network Interface Card (SmartNIC), *e.g.*, Netronome Agilio [47]. These components each have their own strengths and weaknesses. SmartNICs are installed within dedicated servers or hosts on the network and can only enable inference at specific locations in the network. However, they are not highly constrained in terms of available resources and can enable the deployment of more complex models like NNs in the user plane. In contrast, switches are present at multiple locations in the network and thus, can enable ubiquitous in-network inference. Yet, they are highly constrained in terms of available memory and supported operations, limiting the complexity of deployable models. In addition, SmartNICs cost almost as much as programmable switches while offering only 1–4 ports. Switches on the other hand could offer anywhere between 32–64 high-speed ports which can all operate individually at Gbps rates, leading to aggregated data rates in the Tbps range. Ultimately, switches offer better performance, higher data rates, and more ideal location in the network, which are especially important for high-throughput and low-latency user-plane inference. This justifies their adoption in this thesis as the target for user-plane inference.

- *How can the constraints of user-plane equipment be accommodated?*
  All the constraints of programmable user planes must be accommodated in the design of solutions for in-switch inference. As discussed in Section 1.1, most existing works for in-switch inference suffer from performance and scalability barriers. Most of these barriers stem from the inability of those solutions to adapt to the strict constraints of specific user-plane targets. This thesis proposes several techniques and workarounds to accommodate these constraints and design scalable solutions for line-rate inference in programmable switches with DT and RF models.

User-plane inference with ML is a relatively new area of research, with most existing works proposed over the last five years, as recently surveyed [30, 31]. The majority of prior work focuses on classification, used for tasks like network intrusion detection, device identification, and service fingerprinting. This thesis also focuses on classification, which runs fully in the user plane and is analogous to the forwarding role of the switch, which in essence, is the classification of packets to different forwarding ports.

1.3. Contributions

In response to the questions raised in Section 1.2 above, this thesis has made several contributions resulting in peer-reviewed publications and open-source code. Two publications have been published in the proceedings of *IEEE INFOCOM*, which is a tier-1 conference as ranked by the CORE2023[1] database. Other publications include one tier-2 conference paper in *IEEE/IFIP NOMS*, three workshop papers in *ACM e-Energy*, *NativeNI*, and the *NOMS Doctoral Symposium*, and one demo at *IEEE NetSoft*. A journal paper is under review at *IEEE/ACM Transactions on Networking*, indexed in the first quartile of Journal Citation Reports (JCR). The contributions of the thesis are as follows.

Contribution 1. *Packet-level inference use case: cyberattack detection in smart grids*
The first contribution is a practical application of user-plane inference. ML models are widely used for cyberattack detection in Smart Grid (SG) systems based on SDN. These models either run in external servers or in-network but fully in the control plane or distributed between the control and user planes. In all three cases, the models do not run at line rate and incur millisecond-level delays in attack detection. This thesis explores how ML inference in the user plane can enable accelerated attack detection and mitigation in SGs at line rate with sub-microsecond delay. The proposed workflow [3] brings the concept of user-plane inference to SGs and deploys a trained DT model into the switch pipeline for real-time inference on live traffic. Evaluation results reveal how the model can distinguish multiple attacks against SGs with high accuracy, incurring a delay of less than a microsecond while consuming a tiny portion of the available resources in the switch.

[1] https://portal.core.edu.au/conf-ranks/2074/

- Aristide Tanyi-Jong Akem, Michele Gucciardo and Marco Fiore, "Ultra-Low Latency User-Plane Cyberattack Detection in SDN-based Smart Grids". In: *The 15th ACM International Conference on Future and Sustainable Energy Systems (E-Energy '24), June 04–07, 2024, Singapore, Singapore. ACM, New York, NY, USA 7 Pages. https://doi.org/10.1145/3632775.3661995*

Contribution 2. *Hierarchical packet-level inference*
As Packet Level (PL) models like those in Contribution 1 employ only simple header fields as features, they often fail to achieve high accuracy in complex classification tasks, or lead to models that are too resource-hungry to be deployed in the switch. To remedy the situation, this thesis proposes `Henna` [4], a framework for two-stage hierarchical ML inference in switches with DT and RF models. It enables tackling difficult tasks by splitting them into smaller sub-tasks which individually are easier to solve with simpler models that are more adapted to in-switch deployment. Experimental results show that in a device identification task with a large number of classes, `Henna` outperforms a fully grown single-stage classifier, while consuming only modest amounts of switch resources, thereby setting a new standard for solving complex problems with inherent hierarchical relationships between the classes.

- Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo, and Marco Fiore, "Henna: Hierarchical Machine Learning Inference in Programmable Switches". In: *The Proceedings of the 1st International Workshop on Native Network Intelligence (NativeNi'22)*, Roma, Italy, 2022, doi: https://doi.org/10.1145/3565009.3569520.

Contribution 3. *Practical and general-purpose flow-level inference*
Despite the advancements brought about by `Henna`, it remained subject to the accuracy barriers suffered by PL solutions which do not employ any Flow Level (FL) features. As such, there was still no practical solution for running inference at FL in hardware switches with RFs. Prior FL works were either not fully tested in hardware (pForest [41] and SwitchTree [42]), or were use-case specific (pHeavy [44]). In response, this thesis proposes `Flowrest` [2], the first comprehensive framework for deploying FL models into hardware switches, accounting for the constraints of the switches right from the design phase of the models. `Flowrest` is implemented as P4 software and evaluated through multiple experiments which reveal how it outperforms several representative benchmarks. Building on `Flowrest`, a demo [7] is designed to showcase the in-switch ML workflow. In addition, an Encrypted Traffic Classification (ETC) use case is proposed [5], exploiting `Flowrest` but using only features based on packet size and packet Inter-Arrival Time (IAT). This ensures that the models remain robust in the face of encryption since the derived features are typically unaffected by encryption.

- Aristide Tanyi-Jong Akem, Michele Gucciardo and Marco Fiore, "Flowrest: Practical Flow-Level Inference in Programmable Switches with Random Forests". In:

IEEE INFOCOM 2023 - IEEE Conference on Computer Communications, NY, USA, 2023, pp. 1–10, doi: https://doi.org/10.1109/INFOCOM53939.2023.10229100.

- Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo and Marco Fiore, "Showcasing In-Switch Machine Learning Inference". In: *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, Madrid, Spain, 2023, pp. 299-301, doi: https://doi.org/10.1109/NetSoft57336.2023.10175464. **(🏆 Best Demo)**

- Aristide Tanyi-Jong Akem, Guillaume Fraysse and Marco Fiore, "Encrypted Traffic Classification at Line Rate in Programmable Switches with Machine Learning". In: *NOMS 2024 - IEEE/IFIP Network Operations and Management Symposium*, Seoul, South Korea, 6-10 May 2024, pp. 1-9. https://hdl.handle.net/20.500.12761/1791.

- Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo and Marco Fiore, "Practical and General-Purpose Flow-Level Inference with Random Forests in Programmable Switches". In: *IEEE/ACM Transactions on Networking*, 2024. (Submitted)

Contribution 4. *Joint packet-level and flow-level inference*
While PL solutions hit performance barriers in complex tasks, FL solutions improve classification performance but leave the first few packets of flows unclassified during the FL feature computation stage. These missed packets could have diverse effects depending on the use case, including a possible malware infection in security applications. To bridge this gap, prior work proposed a hybrid solution for simultaneous PL and FL inference, using multiple PL and FL models deployed in the switch. However, deploying multiple models greatly increased switch resource consumption. This thesis bridges the gap by proposing `Jewel` [8], a resource-efficient framework for tackling joint PL and FL inference using a single RF model trained to infer on early packets at PL and then switch to FL once computed FL statistics are accurate enough. Experimental results shed light on how `Jewel` consistently outperforms existing state-of-the-art solutions across diverse use cases, while keeping switch resource usage under control.

- Aristide Tanyi-Jong Akem, Beyza Bütün, Michele Gucciardo and Marco Fiore, "Jewel: Resource-Efficient Joint Packet and Flow Level Inference in Programmable Switches". In: *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, Vancouver, Canada, 2024, pp. 1-10. https://hdl.handle.net/20.500.12761/1777.

The publications above which provide details on the contributions of this thesis are accompanied by source code which has been made open-source on GitHub. The list of released source code is presented in Table 1.1. The code is mainly in the form of Python, P4 and Shell scripts which enable the replication of the results of this thesis to improve understanding and appreciation of the proposed solutions, and encourage their adoption.

Name	URL	Paper
Henna	github.com/nds-group/Henna	[4]
Flowrest	github.com/nds-group/Flowrest	[2, 6, 7]
Jewel	github.com/nds-group/Jewel	[8]
Encrypted traffic classification	github.com/nds-group/ETC	[5]
Smart grid cyberattack detection	github.com/nds-group/Smart-Grid	[3]

Table 1.1: Software released as open source.

This provides material to serve as a starting point for new researchers in the area and fills the gap left by most previous works which do not make their source code publicly available and are therefore difficult to replicate.

1.4. Outline of the thesis

The rest of the thesis is organized into different chapters laying out the context within which the solutions proposed in this thesis were developed, and detailing the aforementioned contributions. The chapters are described next.

Chapter 2 lays the foundation of the thesis by presenting the background. It begins by describing the SDN architecture with a programmable control plane and how it has evolved into a fully programmable network with a programmable user plane. It goes further to provide a background on user plane inference, describing the different targets and model deployments that have been proposed to date. Then an evaluation is conducted to assess different ML models for user-plane inference over different use cases. At the end of the evaluation, the focus is then narrowed down to the deployment of DT and RF models for in-switch inference, examining existing works, and exposing their limitations.

In Chapter 3, the methodology of the thesis is presented. All the procedures and tools that are used in the solutions designed in this thesis are outlined. It begins with a description of the user-plane inference workflow including; the model preparation phase which happens in an offline ML server, the mapping of the DT and RF models to the switch pipeline, a description of the control plane components, and finally, how the solutions are embedded into the user plane. Next, the experimental hardware testbed used throughout the thesis is described, followed by the evaluation metrics and datasets used.

The next chapters detail the contributions of the thesis and are divided into three parts. In Part I, there are two chapters which focus on PL solutions for user-plane inference. In Chapter 4, a practical application of user-plane inference is presented. It provides details on the design of a solution for rapid detection of cyberattacks smart grids (SGs). It begins with an introduction to the problem of rapid detection of attacks in SGs, followed by a description of related work and their shortcomings whose solutions are the goal of the chapter. The proposed workflow is then presented and evaluated with experiments on the hardware testbed.

Chapter 5 presents `Henna`, a framework for hierarchical PL inference in the user plane with ML. It begins with an introduction that establishes the need for hierarchical inference as a solution to scalability issues posed by flat classification schemes. Then, the proposed `Henna` framework is presented as a two-stage tree-based model, employing RFs in the first stage and DTs in the second stage. It ends with a thorough experimental evaluation of the framework and a presentation of the results.

Part II comprises two chapters which focus on FL inference in the user plane. Chapter 6 provides details of the design of `Flowrest`, a practical and general-purpose framework for FL inference in programmable switches with RF models. It begins with an introduction that justifies the need for FL inference, and then presents the framework from conception to design.

In Chapter 7, application use cases of FL inference are presented and a thorough experimental evaluation is conducted in two steps. First, `Flowrest` is evaluated on unencrypted traffic over five use cases based on measurement data, comparing it to five benchmarks, also focusing on unencrypted traffic, at different levels of comparison. Then in the second step, the ability of `Flowrest` to classify encrypted traffic is demonstrated by conducting an experimental evaluation with three use cases. It ends with the presentation of a demo framework designed to showcase in-switch inference with a dashboard that displays the results.

Part III contains only Chapter 8 which is about joint PL and FL inference in the user plane. It presents `Jewel`, a framework for joint PL and FL inference in programmable switches. It begins with a motivation of why joint inference is needed in the face of the limitations of existing PL, FL, or hybrid solutions, and then goes on to present the details of the joint inference model from the design phase to the implementation phase, followed by a detailed evaluation over several use cases, in comparison to various benchmarks.

Chapter 9 summarizes the findings and conclusions of the thesis. It then lays out possible directions for immediate future research, followed by a broad vision of the research area over the next decade.

2 Background

The journey towards user-plane inference has lasted over three decades, beginning with research on active networks, advancing to the proposal of the Software Defined Networking (SDN) concept, and then programmable user planes and domain-specific network programming languages, which have made the network fully programmable.

2.1. The road to programmable networks

By the early 90s, network innovation seemed to have been impaired by an apparent ossification of network infrastructure and protocols [48]. Ossification refers to the impairment of network evolution resulting from the fact that networks were typically designed to support a defined set of services, such that enabling support for a new set of services required modifying the functioning of each node and a group of middleboxes which typically had individual control components as shown in the traditional network architecture in Figure 2.1. This greatly hampered network evolution and flexibility.

Attempts to make the network more evolution-friendly in the early 90s sparked an interest in research on *active networks* [49]. These networks allow their users to inject customized programs into network nodes [50], thereby enabling innovation. In the early 2000s, the concept of SDN became much clearer with the advent of the separation in the network of the *control plane* from the *user plane* as depicted in Figure 2.1. Conventionally, the control plane determines how data is moved through the network, *i.e.*, by setting routing and forwarding policies, while the user plane executes forwarding decisions and moves network traffic around. However, in recent years, more sophisticated applications are being deployed in the control plane for network automation. The SDN architecture also came with open interfaces between the two planes and provided a global view of the network from a central management point in the control plane.

With the coming of the OpenFlow API [51] around 2008, alongside network operating systems, *e.g.*, NOX [52], network programmability became more popular - *albeit in the control plane*. OpenFlow provides a means to control user-plane switches using flow

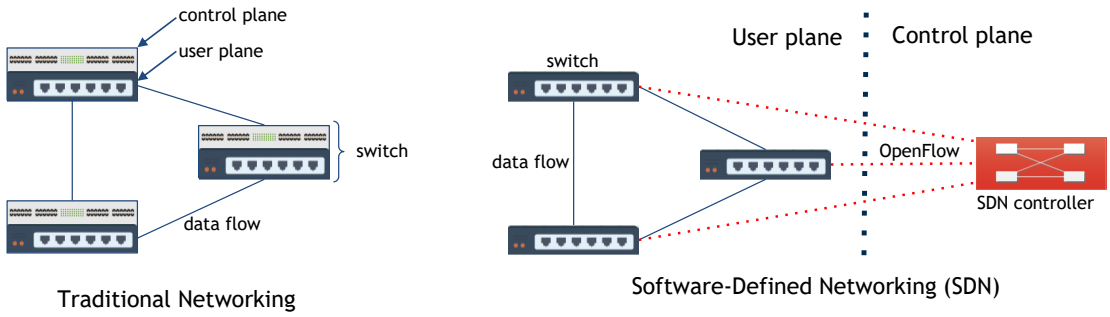


Figure 2.1: Overview of traditional and software-defined network architectures.

table entries that are supplied by a remote controller communicating with the OpenFlow switches via a secure channel using the OpenFlow protocol [51]. Although OpenFlow greatly improved network programmability and flexibility, as the protocol evolved, it had to be able to operate on more and more header fields. This led to an increasing complexity of the protocol specification, without providing the possibility to add new headers.

To evolve OpenFlow and extend programmability to the user plane, the *Programming Protocol-Independent Packet Processors* (P4) language was proposed in 2013 [25]. P4 is a high-level programming language for writing programs that tell packet processors how to process packets. It enables reconfigurability in the field, protocol independence and target independence; which extend network flexibility and programmability, rendering the network fully programmable. This brings several benefits, *e.g.*, improved efficiency in memory usage since programs are written to only parse required headers, greater control over the entire network since both planes can be customized, and the emergence of several new applications of SDN [53] in general, or of user-plane programmability [26, 27].

2.2. In-network inference

Modern networks are increasingly being required to grow in complexity to support emerging applications that require ultra-low latency, *e.g.*, autonomous driving [22], augmented and virtual reality [21], and the Metaverse [54]. As such, there is an increased need for automation, enabled by data-driven solutions that can improve network autonomy and support for complex applications. Machine Learning (ML) algorithms are playing a key role in network automation by enabling several intelligent applications [55].

In the context of conventional SDN networks, ML models run in the control plane and enable numerous applications, *e.g.*, network planning, resource allocation, fault detection, routing optimization, and anomaly detection, as reviewed in multiple surveys [56–58]. These applications enable ML-based inference, by which a model running in the network control plane can take decisions which are then used to design informed policies for the network. Most in-network inference applications perform classification tasks, through

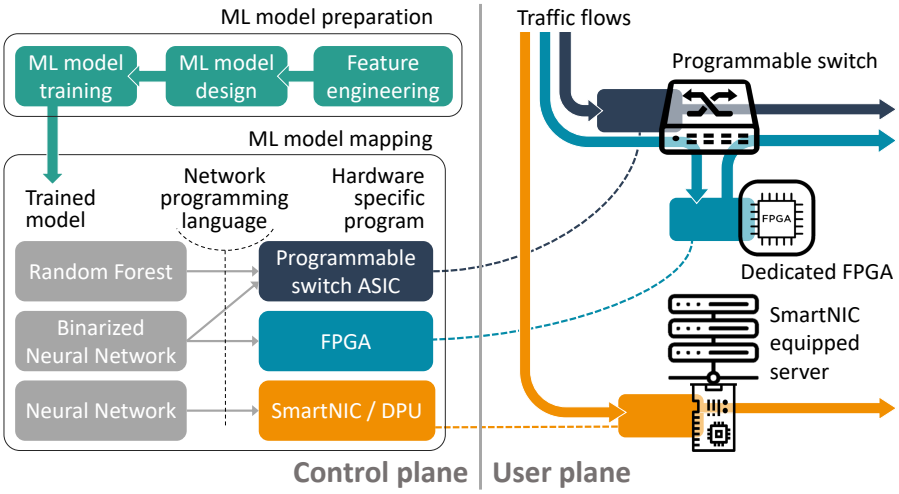


Figure 2.2: Summary of the approaches for user-plane inference.

which network packets, flows or sessions are categorized into various groups. This is intuitive considering that the traditional role of the network is traffic forwarding, which is analogous to classifying packets to different output ports.

While these applications greatly enhance network automation, many do not operate in real-time on live network traffic. Even when they do, they do not run at line rate, *i.e.*, *at the same rate at which data is forwarded*, and incur anything from one to hundreds of milliseconds of delay [19], which is undesirable for ultra-low latency applications. To reduce the delay and move closer to line rate, proposals have been made to distribute ML tasks between planes, so that by alleviating the control plane of some tasks, the user plane can assist in, *e.g.*, data aggregation [29, 59, 60], or feature extraction [32, 33]. Yet even in these cases, the inference process does not occur fully in the user plane hence, line rate remains unattainable, leaving room for improvement in the area.

2.3. Line-rate inference in the user plane

Building on the recent advancements in user-plane programmability, deploying ML models in the user plane promises to fill the gaps left by control-plane solutions and enable line-rate inference with high throughput and low latency. However, embedding models into the user plane is an uphill task, owing to the strict constraints discussed in Chapter 1. As a result, user-plane inference applications have focused on deploying trained models into the user plane for line-rate inference on live traffic, as described next.

This section provides a brief but complete review of existing solutions for line-rate inference in the user plane. As this thesis focuses on pure user-plane implementations of ML models, this review does not consider hybrid strategies that split the inference load across planes [29, 59–61], nor does dwell on criticisms to such strategies [62]. Figure 2.2 offers a unifying view of the existing workflows adopted for line-rate inference.

Given the limitations of programmable network hardware, all existing approaches in the scope of this review assume that all the model preparation phases, including computationally expensive phases such as model hyper-parameter optimization and model training, are performed offline in the control plane, typically using dedicated ML servers with Graphics Processing Units (GPUs) or Tensor Processing Units (TPUs). Trained models are then encoded for operation in the target user-plane equipment via a network programming language, *e.g.*, P4. Proposals vary in terms of (*i*) the family of ML models considered for user-plane implementation, and (*ii*) the nature of the target programmable hardware. Next, prior works are reviewed based on the hardware they target, while also discussing the specific models they implement. A more extensive discussion of these works can be found in two recent surveys on the topic [30, 31].

2.3.1. Programmable switch ASICs

Most user-plane ML deployment efforts have focused on in-switch inference. In this case, as shown in Figure 2.2, the ML model is fully deployed in a single programmable switch, by implementing it into the switch Application-Specific Integrated Circuit (ASIC): this allows processing packets at line-rate using solely the resources of the switch. Here, Decision Tree (DT) and Random Forest (RF) models are common choices, due to their relatively low complexity and intuitive mapping to the ASIC pipeline. DTs and RFs are in fact semantically equivalent, since RFs are realized by running multiple DTs in parallel. As this is the strategy adopted for the solutions proposed in this thesis, an in-depth discussion of current in-switch solutions with DTs and RFs is presented in Section 2.5.

Other ML models have been also tested for in-switch operation. The list includes Support Vector Machines (SVM), Naive Bayes, K-Means, XGBoost, or Isolation Forest; however, DTs and RFs were ultimately found to offer higher scalability and better performance [38, 40]. The same holds for attempts targeting more complex models, such as Neural Networks (NN) [40, 63], possibly with a Binarized flavour (BNN) [36] that relies on $+1/-1$ weights and sign activations [37]. Even simplified versions of NNs are onerous to deploy into commodity programmable switches, *e.g.*, a very basic Binarized Neural Networks (BNN) with two layers of 64 and 32 neurons completely exhausts the resources of an Intel Tofino ASIC [36], while yielding poor inference results due to its exceeding simplicity. Also, as it will be shown in Section 2.4, these complex models are often outperformed by simple DT and RF models in different use cases.

A few recent works have engineered techniques to deploy NN-based models on switches. Some rely on distilling the model into a simple decision tree that can run on the switch [64]. However, distillation often leads to huge accuracy losses when tackling complex tasks. Other works propose more compression and quantization techniques for encoding NN models into switch pipelines [65, 66]. Yet, these solutions achieve similar or lower accuracy when compared to state-of-the-art DT or RF-based in-switch solutions. More recent

works like RIDS [67] and BoS [68] have proposed complex encoding techniques to embed recurrent neural networks (RNNs) on switches. While these models achieve promising results on the use cases tested, they only use $1-2$ features, *i.e.*, the sequence of packet sizes and interarrival times which would not suffice to handle complex classification tasks requiring additional features from other protocol headers [2].

2.3.2. Smart Network Interface Cards (SmartNICs)

The fundamental limits of programmable switches in supporting NNs have fostered the exploration of alternative user-plane hardware to accommodate such models. N3IC [69] presents a comprehensive approach for the integration of BNNs on a Smart Network Interface Card (SmartNIC), using both the micro-C language (for Netronome system-on-chip NICs) and P4 (for NetFPGA NICs configured with a PISA architecture). Figure 2.2 highlights (in orange) how N3IC maps the ML model to SmartNIC hardware located in a server. Incoming traffic can then be classified in real time by the deployed model. The SmartNIC environment alleviates the constraints of programmable switches, enabling N3IC to implement a 3-layer BNN that operates at line rate while consuming only a relatively small fraction of the available hardware resources. This avoids that the ML inference is performed by the CPU/GPU of the host (which introduces delay) and that it instead occurs entirely in the NIC (at line rate).

Yet, SmartNICs are deployed at network appliances (*e.g.*, traffic classifiers, load balancers, or security middleboxes) that physically reside in a dedicated host within the network datacenter. Approaches like N3IC can thus grant line-rate inference at specific locations of the network only, and not at any point of the transport domain as it can potentially happen with in-switch solutions. In other words, malicious traffic classification could be performed at some specific point of the network (where the dedicated host is located) but not at any point of the network (*i.e.*, at any switch). It is also noteworthy that a single SmartNIC is priced similarly to a production-level programmable switch while supporting a much smaller number of ports (1–4 versus 16–64), and is thus substantially less cost-efficient on a per-port basis [70].

2.3.3. FPGA-enhanced switches

An even more radical strategy is adding dedicated FPGAs to switches or SmartNICs and exploiting them to implement complex NN models. The Taurus [71] framework employs a custom accelerator based on a pipelined Single Instruction Multiple Data (SIMD) parallelism to implement NNs via MapReduce operations. Taurus-enhanced switches offload to the external hardware, the per-packet inference process, enabling the deployment of powerful ML models in user planes. The obvious shortcoming of the approach taken by Taurus lies in substantial added costs and technical complexity. Its

adoption at scale would require revisiting the user plane design and deploying significant custom hardware next to already expensive programmable switches and SmartNICs. This would also be true for the approach proposed by Monterubbiano *et al.* [72] for representing FPGA-based RF models which is evaluated against state-of-the-art representations and shown to be efficient in terms of computation and memory requirements. These approaches are therefore not adopted in this thesis which instead focuses on deploying models into the user plane without any structural modifications or the addition of new components *e.g.*, FPGAs.

2.4. Comparative evaluation of user-plane ML models

The different studies on user-plane ML listed in Section 2.3 present different model types targeting assorted use cases and diverse traffic datasets [36, 38, 41–43, 71]. This demonstrates the potential of multiple model types to be deployed in programmable user planes. However, owing to the varying complexity of these model types and the strict constraints of user-plane environments, it is important to assess the performance of the different kinds of models over varying use cases to determine what models will offer the best accuracy-complexity trade-off. Such models are most appropriate for user-plane deployment. No prior work conducts such an analysis and this thesis fills that gap.

2.4.1. ML models

In this section, an evaluation is conducted in Python to test the performance of DT, RF, NN and BNN models parametrized according to the indications in prior works [69]. The model configurations are briefly described next.

Decision Tree (DT). Binary DTs are considered with depths from 3 to 10 depending on the use case. Model sizes are based on the evaluation conducted in N3IC [69].

Random Forest (RF). RFs of 5 binary trees with depth from 3 to 10 are implemented depending on the use case [69]. The DTs and RFs are trained and evaluated using the open-source Scikit-Learn [73] library used by all works relying on these types of models for user-plane inference [38, 41–43].

Neural Network (NN). NNs based on multi-layer perception (MLP) are used with one or two hidden layers of 3, 6, 32 or 64 neurons; the input and output layers have 12, 64 or 128 neurons and 2, 6 or 8 neurons, respectively depending on the use case [69, 74]. NN models are trained and evaluated using the popular open-source Tensorflow [75] libraries.

Binarized Neural Network (BNN). Binarized versions of the NNs above are evaluated with the same architecture and hyperparameters, but with binary weights and activations [37, 76]. To implement BNN models, Larq [77] is used. It is an open-source Python library for the training of neural networks with quantized weights and activations, based on state-of-the-art algorithms [37].

Dataset	Year	Flows	Features	Classes
UNSW-IoT Traces [78]	2016	423,491	18	6 (audio, gateway, other, sensor, static, video)
UNIBS Internet Traces [79,80]	2009	40,338	18	8 (bittorrent, edonkey, http, imap, pop3, skype, smtp, ssl)
CICIDS-2017 [81]	2017	306,775	80	2 (attack, normal)
UNSW-NB15 [82,83]	2015	306,799	49	2 (attack, normal)
NSL-KDD [84]	2009	148,517	42	2 (attack, normal)

Table 2.1: Summary of the datasets considered in the comparative analysis.

In addition, other configurations of the hyperparamenters of each model were tested, by varying the number of trees in RFs, or the number of layers and neurons per layer in NNs and BNNs. The experiments confirm the validity of the original settings proposed in the literature for the different use cases, which either grant the best performance or require minor changes (*e.g.*, adding one tree to the RF model) in each task.

2.4.2. Use cases

To ensure maximum fairness of the evaluation, we produce results for the majority of use cases explored in the original works presenting each inference model for the first time [36,38,41–43,69,71]. Only use cases for which the source traffic data is not publicly available are excluded. This results in two traffic classification tasks and three intrusion detection tasks, which rely on the datasets summarized in Table 2.1. Those that are re-used in other chapters are only briefly described here and then fully in Chapter 3.

UNSW-IoT [78] is a classification use case based on measurement data for 28 Internet-of-Things (IoT) and non-IoT devices. More details are provided in Chapter 3. The objective is to classify traffic into one of 6 classes depicting the kind of IoT device generating the traffic. Models are trained over the first 15 days and tested on the last 5.

UNIBS-2009 [79,80] is a traffic classification task based on a 3-day real-world trace. The goal is to classify traffic flows into one of the 8 application categories listed in Table 2.1. One day of traffic is used for training and a second for testing.

CICIDS2017 [81] is an intrusion detection use case. It is detailed in Chapter 3. The Friday dataset is used and the objective is to identify all malicious flows. A $75-25$ train-test split ratio is used when training and testing models.

NSL-KDD [84] is an intrusion detection case study that builds on 7 weeks of network traffic captured on a testbed recreating normal and attack traffic behaviours, by exploiting real hosts, live attacks and background traffic. The attacks fall in one of the following 4 categories: Denial of Service Attack (DoS), User to Root Attack (U2R), Remote to Local Attack (R2L), and Probing Attack. The goal is also to separate malicious and regular traffic. The full data is considered with the default training and testing separation.

UNSW-NB15 [82,83] is an anomaly detection task employing a mix of real-world normal traffic and concurrent synthetic attack behaviours, produced in the Cyber Range Lab of UNSW Canberra. The measurements sum up to 100 GB of raw traffic with nine different types of attacks The goal is once more to identify all malicious flows. In

experiments, 10 GB of data from the second day is extracted and 5 GB is used for model training, while the remaining 5 GB is used for testing.

2.4.3. Metrics

The performance of each model across the different use cases is assessed via a wide range of metrics that cover those used in the original studies listed in Section 2.3. The metrics are defined in Chapter 3 and include accuracy, precision, recall and F1-score. For every metric, the final value is averaged over all classes.

For completeness, two sets of results are reported, *i.e.*, for full and early flows. Full flows imply that the statistical features associated with a traffic flow are computed by observing its complete set of packets, and have been considered in the evaluation of N3IC [69] and SwitchTree [42]. Early flows are used to assess pForest [41], and enact that inference is completed over the first few packets of each flow only, which reportedly improves the speed of response without sacrificing quality [85, 86]. In the conducted experiments, the first 5 packets are used when tackling the UNIBS-2009 task, and the first 3 packets are used in all other use cases, which is aligned with the settings of the pForest evaluation [41]. Early flow tests could not be run for NSL-KDD, which lacks the raw capture data required to extract the first packets of a flow.

Also, the packet-level inference adopted in the works introducing IIsy [38], Planter [43] or Taurus [71] is not adopted. The rationale is that running inference on each packet separately does not allow gathering flow-level features (*e.g.*, packet inter-arrival times) that prove key to the quality of the result as will be demonstrated in Part II. For instance, packet-level intrusion detection yields a 71.1 F1-score in NSL-KDD [71], versus the 97.07 value obtained with the best flow-based model.

2.4.4. Results

The results are summarized in Table 2.2, and are quite manifest. The DT, RF and NN models achieve performance that can be considered satisfactory across all use cases, with F1-scores typically in the $95 - 100$ range, and consistently good in all other metrics. BNNs instead lag with less consistency and lower accuracy: the gap from the second-worst model varies between 10 and 20 points in terms of F1-score. This result can be expected given the reduced flexibility of binary weights: the paper originally introducing BNNs reports 8% performance reduction in MINST classification and 24–33% lower accuracy in more demanding tasks [87], and complications such as multiple binary weight bases are required to improve the latter to 5–8% [88]. Such complications would render these models harder to deploy into user-plane targets due to the inherent constraints.

A second takeaway is that all metrics show nearly identical values under full and early flow approaches. Since computing features on early flows grants anticipated

			Accuracy		Precision		Recall		F1-score	
Use case	**Dataset**	**Model**	**Full flows**	**Early flows**	**Full flows**	**Early flows**	**Full flows**	**Early flows**	**Full flows**	**Early flows**
Traffic Classification	UNSW-IoT	DT(10)	96.254	97.771	96.617	97.717	95.292	97.628	95.896	97.645
		RF(10,5)	**98.758**	**98.695**	**98.776**	**98.984**	**98.499**	**98.593**	**98.635**	**98.779**
		NN(128,64,6)	97.249	97.362	97.336	97.361	97.249	97.362	97.249	97.338
		BNN(128,64,6)	72.267	83.002	74.481	84.435	72.267	83.002	70.858	82.909
	UNIBS-2009	DT(9)	99.359	**99.737**	93.499	**96.345**	**93.931**	**97.406**	93.706	**96.851**
		RF(9,5)	**99.532**	99.521	**96.091**	95.077	92.955	95.107	**94.280**	95.078
		NN(64,32,8)	97.838	99.223	90.965	95.261	92.471	89.116	91.627	91.129
		BNN(64,32,8)	85.501	89.747	80.063	90.479	85.501	89.746	80.792	87.169
Anomaly Detection	CICIDS-2017	DT(10)	**99.915**	99.744	**99.924**	99.799	99.824	99.590	**99.874**	99.694
		RF(10,5)	**99.915**	**99.762**	**99.924**	**99.805**	99.824	**99.628**	**99.874**	**99.716**
		NN(128,64,2)	99.865	99.259	99.691	99.263	**99.909**	99.259	99.799	99.260
		BNN(128,64,2)	97.748	89.807	95.224	87.305	98.563	92.622	96.765	88.802
	UNSW-NB15	DT(7)	98.712	98.883	**86.768**	90.177	94.957	92.467	90.417	91.288
		RF(10,5)	**98.772**	**99.213**	86.398	**91.492**	**97.655**	97.055	**91.206**	**94.084**
		NN(128,64,2)	98.533	98.971	85.628	88.056	93.255	**99.029**	89.040	92.789
		BNN(128,64,2)	94.284	91.587	64.236	62.405	78.293	87.599	68.528	67.214
	NSL-KDD	DT(10)	**97.037**	–	**97.015**	–	**97.071**	–	**97.035**	–
		RF(10,5)	96.990	–	96.973	–	97.004	–	96.986	–
		NN(12,6,3,2)	91.984	–	92.275	–	91.830	–	91.939	–
		BNN(12,6,3,2)	85.254	–	87.406	–	84.770	–	84.895	–

Table 2.2: Results of the comparative evaluation of machine learning models used for user-plane inference, across the five target use cases. The best result for each use case and metric is highlighted in bold, and the second best in pink.

classification of traffic or detection of anomalies, it is largely preferable in all considered use cases. Thirdly, and most importantly, RF emerges as the overall best model from the comparative evaluation. The model is often the best performing one or is a close second otherwise. The metrics are definitely on par with (and often better than) those attained by the more complex NN architecture. This is an important observation, since, by looking at Section 2.3, NN models require simplifications (*e.g.*, via binarization) or additional components (*e.g.*, dedicated FPGA accelerators) to be implemented in the user plane. Simpler RF models do not suffer from these limitations, yet achieve equivalent or superior inference performance in all target use cases.

In summary, the comparative analysis shows that *RF models based on early-flow detection* are a promising candidate for deployment of ML in the user plane, assuming that they can be efficiently integrated into off-the-shelf programmable hardware. The same can be said of DT models which also come first or second in multiple cases. This conclusion is not entirely surprising since prior work has also demonstrated that tree-based models often outperform deep learning in tabular data tasks [46]. The above evaluation is a strong motivation for the choice of in-switch DT and RF models in this thesis.

2.5. In-switch tree-based inference

This section emphasizes in-switch inference using DT and RF models. As explained in Section 2.3.1, executing ML models on standalone programmable switch ASICs allows for the most pervasive user-plane deployment without incurring additional hardware costs; and, based on their simplicity and ability to outperform more complex models, tree-based models stand out as the most effective solutions in these highly constrained environments. Another motivation for the choice of DT and RF models is that their logical structure is

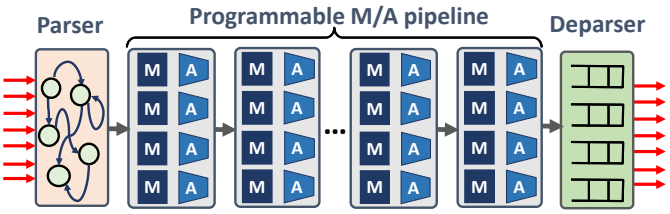


Figure 2.3: Protocol Independent Switch Architecture (PISA).

Solution	Design			Scalability		Practicality			
	HW-tailored configuration	P-L support	F-L support	Forest support	Unrestricted tree depth	General purpose	Hardware implementation	Resource usage analysis	Open code
Ilsy [38, 39, 89]		✓		✓	✓	✓	✓		
pForest [41]			✓	✓		✓	✓		
SwitchTree [42]			✓	✓		✓			✓
Planter [40, 43, 90]		✓		✓	✓	✓	✓		
NERDS [91, 92]			✓			✓			✓
pHeavy [44]			✓				✓		
Mousika [64, 93]		✓			✓	✓	✓	✓	✓
Akem et al. [3]		✓			✓		✓	✓	✓
BACKORDERS [94]			✓	✓					
Netpixel [95]		✓			✓				✓
Henna [4]		✓		✓	✓	✓	✓	✓	✓
Bütün et al. [10]		✓		✓	✓		✓	✓	✓
Soter [96]		✓					✓		✓
INC [97]			✓		✓		✓		
Friday et al. [98]			✓	✓	✓		✓		
Flowrest [2, 6]	✓		✓	✓	✓	✓	✓	✓	✓
Genos [99]			✓		✓		✓	✓	✓
Leo [100]			✓			✓	✓	✓	✓
NetBeacon [101]		✓	✓	✓	✓	✓	✓	✓	✓
BoS [68]		✓	✓	✓	✓	✓	✓	✓	✓
Jewel [8]	✓	✓	✓	✓	✓	✓	✓	✓	✓

Table 2.3: Comparative summary of solutions for in-switch inference and with DT and RF models. Columns refer to (*i*) adoption of ML modelling and experiment configuration approaches that are tailored to the switch hardware requirements by design, (*ii*) support for packet-level inference, (*iii*) support for flow-level inference, (*iv*) support for RFs composed of multiple DTs, (*v*) lack of structural constraints to the depth of the trees, (*vi*) applicability to general inference problems opposed to solving a dedicated task only, (*vii*) implementation and experimental evaluation with a real-world hardware platform, (*viii*) complete analysis of switch resource consumption based on memory types, (*ix*) availability of open-source code. Solutions proposed by this thesis are highlighted in pink.

analogous to the Match & Action (M/A) pipeline of the switch which is generally based on the Protocol Independent Switch Architecture (PISA) shown in Figure 2.3. The parser is analogous to the feature extraction/computation phase which happens before the model is run. Then, each M/A stage is analogous to a tree node or stage, where the "match" is the comparison between the threshold and the feature, and the "action" is the selection of a tree branch to follow. Lastly, the arrival of a packet at the deparser after classification is analogous to a sample arriving at a tree leaf.

Table 2.3 summarizes major studies exploring in-switch tree-based inference, categorized in three dimensions: design methodology, scalability to complex models and use cases, and suitability for practical environments. Next, the specifics of prior solutions are discussed and compared to those of the solutions proposed in this thesis.

2.5.1. Design

Regarding model design, the main dichotomy is between performing Packet Level (PL) or Flow Level (FL) inference. In PL design, models are fed with features extracted independently from the headers of each packet. The PL approach makes implementation easier, as feature extraction can be directly realized via the native header parsing functionalities of programmable switches. Yet, it also has notable drawbacks. First, PL models lack access to features calculated over multiple packets of the flow, such as Inter-Arrival Time (IAT) or packet size statistics, which play an important role in accurately solving complex network problems [102–106]. Second, PL models must be executed on every packet traversing the switch, which may incur higher energy consumption due to the more frequent access to power-hungry Ternary Content-Addressable Memory (TCAM) [107].

Most existing works for PL inference focus on flat classification, in which a single model is trained to tackle each classification task, irrespective of its size and complexity [3, 10, 38, 40, 93, 95, 96]. This often leads to very large models that are not feasible for user-plane implementation. This thesis addresses this issue by proposing `Henna` [4], the first model for hierarchical inference in the user plane. `Henna` avoids training very complex models by instead splitting classification tasks into simpler cascaded problems that can each be solved with a smaller model. These simpler models are then deployed for line-rate in-switch inference in stages, as detailed in Chapter 5.

On the other side of the divide, FL inference utilizes multi-packet features computed over the first few packets of a flow and then applies the result to all subsequent packets of the same flow. It thus has the potential to remove the limitations of PL solutions by (*i*) enabling the use of FL statistics and (*ii*) making decisions that apply to all packets of a flow while processing only a few of them. Nevertheless, the FL approach also introduces substantial realization challenges, such as the need to store stateful FL features or to manage classified flows. As a result, and despite its advantages, only a subset of the solutions in the literature support FL functionalities, as shown in Table 2.3.

This thesis advances the state of the art by proposing `Flowrest`, the first complete practical and general-purpose framework for integrating FL ML models into commercial programmable switch ASICs, as detailed in Chapter 6. A distinctive feature of `Flowrest` is that it puts forth and employs guidelines for designing RFs to guarantee the compatibility of trained models with off-the-shelf programmable equipment. pForest [41] and SwitchTree [42] incorporated feature selection into their operation, but it functions as a traditional model preparation stage and is entirely indifferent to in-switch requirements. Although pForest [41] introduced a feature compression method to conserve switch resources, its practical viability is not demonstrated[1]. Friday *et al.* [98] also attempt

[1]The proposed bit-level compression of features in pForest [41] enables the conservation of a few bits in the representation of each feature. However, it is worth noting that commercial programmable

to consider the constraints of programmable switches in the RF modelling process. They employ Gray coding to feature table codes, enabling the reduction of TCAM consumption. However, this is more of a post-training optimization, performed after the RF model has been generated. In contrast to prior proposals, `Flowrest` is the first to customize feature engineering and hyper-parameter tuning specifically for the underlying user-plane operation, as elaborated in Chapter 6.

Another aspect to consider is the fact that most existing FL solutions have their model mapping coupled with the use case or flow management strategies employed [97,98]. This makes it difficult to exploit those solutions for other use cases or with other RF mapping schemes. `Flowrest` takes a different approach by providing a generic framework that is not tied to any use case and supports any feasible model mapping in a plug-and-play manner, as explained in Chapter 7.

Table 2.3 also reveals that most existing solutions either perform PL or FL inference but not both. PL solutions classify all packets but hit performance barriers in complex tasks. FL solutions on the other hand are unable to classify all packets since they miss the first few packets of flows which they require to compute stateful FL features that they employ to attain higher accuracies. These downsides of both kinds of solutions necessitate joint inference in which both PL and FL inference can be achieved simultaneously, taking advantage of the benefits of both PL and FL inference.

The only prior work on joint inference that is fully based on DT and RF models is NetBeacon [101] which takes a step ahead of existing FL solutions by implementing multiple sequential DT models that enable the classification of either packets or flows at different phases of their life cycle. NetBeacon prioritizes the classification of all packets, rather than full flows and in cases where most flows are short, most of the packets will be classified with the generally less accurate PL model leading to an overall lower accuracy. In addition, deploying multiple models in the switch leads to increased resource consumption for NetBeacon as shown in Chapter 7, without necessarily leading to higher accuracy. Also, although the paper claims to map several RF models for in-switch inference, it is unclear how this is possible and the released artifacts only show how DTs are mapped. Instead, this thesis proposes `Jewel` [8] which performs both FL and PL inference, achieving high accuracy, and using a single model trained to classify both packets and flows in contrast to NetBeacon which employs multiple models.

Another recent proposal for joint inference is BoS [68] which proposes a three-component framework for in-network inference. A user-plane-friendly design of an RNN is proposed for line-rate in-switch inference. It is complemented with a concurrent deployment of a NetBeacon-based model which classifies packets for which state could not be stored due to memory unavailability or during feature computation. If a flow

switch ASICs generally only support byte-level memory allocation, rendering most bit-level optimizations unfeasible in hardware.

cannot be classified accurately by both in-switch models, it is delegated to an offline transformer-based module for further analysis.

The major advantages of `Jewel` [8] over BoS [68] include the facts that `Jewel` employs only a single model and runs fully in-switch without any offload of flows to a control plane model. Thus, `Jewel` consumes less switch resources and incurs lower latency in traffic classification while using only RF models which are typically more explainable. In addition, `Jewel` can choose from over 30 PL and FL features in the switch whereas BoS only relies on two features, packet size and interarrival time. Overall, `Jewel` [8] offers more flexibility and is tailored to switch constraints right from the design phase.

2.5.2. Scalability

Scalability is examined from two angles: (*i*) support for RF models with multiple trees, and (*ii*) maximum supported depth of RF trees.

Regarding RF support, the majority of solutions facilitate multi-tree RF designs. However, recent approaches like NERDS [91], Mousika [93], Soter [96], pHeavy [44], and Leo [100], are constrained to single-tree models, *e.g.*, DTs or Binary DTs (BDT). This limitation is a notable drawback, as RFs, which generalize DTs, are widely acknowledged to possess superior learning capabilities, particularly in more challenging tasks.

Existing solutions utilize various methods to map DTs or RFs into the PISA pipeline employed by most modern programmable user planes. The majority of these methods impose structural constraints on the maximum attainable tree depth. For example, the mapping strategy employed by pForest [41], later adopted by SwitchTree [42] and in a modified form by BACKORDERS [94], maps each tree level to one M/A stage of the PISA pipeline. Thus, the maximum tree depth is limited by the relatively small number[2] of M/A stages in commercial switch ASICs.

Soter [96] also suffers from this limitation as it again employs 1 M/A stage per level of the tree, differing from pForest [41] only by the use of TCAM range matches to compare feature thresholds. In addition, NERDS [91, 92] and pHeavy [44] use mappings in which conditional statements and/or distinct trees are linked to individual M/A stages and need to be executed sequentially. This sequential operation across a restricted number of stages imposes a natural limitation on the model complexity. Mousika [64, 93] takes a more radical approach by generating a binary DT (BDT) from the original model and mapping only the BDT to the switch pipeline. While this brings about a more compact representation of the model, binarization often results in a loss in accuracy as shown in Chapter 7, where a fully-grown BDT trails its RF counterpart by double digits.

[2]In Intel Tofino switches, first-generation chips have 12 M/A stages, while second-generation chips have 20 M/A stages. In the context of FL inference, certain stages need to be allocated for computing flow identifiers and register indices, handling stateful features, and implementing tree leaves. This allocation restricts the maximum achievable tree depth. For example, pForest [41] asserts that only relatively shallow trees with a depth of 4 can be implemented in such hardware.

Genos [99] proposes an in-switch framework for unsupervised FL anomaly-based network intrusion detection by rule extraction. It can analyze a given dataset and generate simple tree-based rules for in-switch inference. The rule-based approach is independent of the depth of the tree, is adapted to the switch pipeline, and is memory efficient. However, Genos does not support RFs. Leo [100] is an online model for line-rate classification of traffic flows in the switch using a DT model. While it achieves high accuracy in the tasks considered, Leo also does not support RFs and the maximum depth of trees is constrained.

Other approaches employ tree encodings that dissociate tree levels from M/A stages, thereby eliminating the systemic constraints mentioned earlier. Notably, the mapping approach initially introduced by Ilsy [38] and subsequently extended by Planter [43] represents the state of the art, as it allows integrating in real-world hardware multiple trees with a depth which is only constrained by the switch memory. In fact, most recent works either re-use this mapping scheme [4, 10, 95] or design new strategies closely related to the former [97, 98, 101]. Even so, the strategies proposed in INC [97] and later extended in Friday *et al.* [98] are specifically tailored to binary classification with only leaf nodes with a positive class result encoded. This limits their application to only binary classification use cases. Hence, while `Flowrest` and the other solutions proposed in this thesis have the flexibility to adopt various methods for mapping RF models into PISA pipelines, they currently use a custom implementation of the mapping approach proposed by Planter [43], as detailed in Chapter 3, to guarantee scalable multi-tree support.

2.5.3. Practicality

Either software or hardware targets can be employed for in-switch inference implementations using the same network programming language, such as P4. When the solution is developed for software, it can only be assessed through emulation, *e.g.*, by executing it on the widely used *BMv2* target within a Mininet environment. While valuable for initial development and debugging, software implementations deviate significantly from production-level hardware targets, such as Intel Tofino ASICs. Disparities extend beyond differences in throughput and latency; emulation masks several challenging constraints inherent in real-world equipment. Hence, solutions exclusively tested in software often lack compatibility with tangible programmable switches, necessitating substantial re-design efforts when porting them, resulting in performance losses [108].

Hence, the primary characteristic determining the practical feasibility of a model is its potential to be implemented in real-world hardware. According to Table 2.2, this applies to the latest versions of IIsy [39] and Planter [40], along with Soter [96], and Bütün *et al.* [10], all of which only support PL inference. INC [97] and Friday *et al.* [98] also have hardware implementations but are only adapted for binary classification and are not directly extensible to general-purpose multi-class inference. Other recent contributions,

such as pHeavy [44], Mousika [93], Leo [100] and NetBeacon [101], showcase hardware implementations, albeit limited to DTs, BDTs or multiple non-RF DTs. Genos [99] also has a hardware implementation but is not general purpose. Also, it is noteworthy that not all the existing solutions make their source code publicly available.

Finally, two more facets are highlighted in Table 2.3. First, most of the proposed models are versatile and applicable to diverse inference tasks, however, pHeavy [44] is dedicated to a specific objective; the binary identification of heavy flows; INC [97] and Friday *et al.* [98] are also specifically dedicated to botnet propagation and ransomware attack detection, respectively, while several others are dedicated only to attack detection and classification [3, 10, 94, 96, 99]. Also, not all prior works explicitly outline the requirements of the models in terms of switch resource usage.

This thesis closes these gaps by introducing novel in-switch inference solutions; notably `Henna` [4], `Flowrest` [2], and `Jewel` [8] which are all general-purpose and are designed to function on production-level hardware while supporting PL, FL, or both types of features, and tackling multi-class problems with multi-tree models. Their evaluation also provides details about the resource usage of PL, FL, and joint PL+FL models on production-level equipment. These solutions set new standards in the area and to enable and encourage verifiability and reproducibility of the proposed solutions, as well as to advance research on in-switch inference, all the proposed solutions are made open source.

3 Experimental methodology

As discussed in Chapter 2, this thesis focuses on the deployment of Decision Tree (DT) and Random Forest (RF) models in programmable switches due to the simplicity of these models that makes them more adaptable to in-switch operation, and the potential of switches to enable high-speed inference anywhere they are located in the network. DTs and RFs only require simple logical comparisons of feature values to the thresholds at tree nodes. They are therefore the most suitable for highly constrained environments like programmable switches. This chapter describes the methodology employed to deploy Machine Learning (ML) models in programmable switches to enable inference at line rate with high throughput and low latency. The adopted workflow is described in Section 3.1. Then, a description of the experimental hardware tested used is presented in Section 3.2, followed by the metrics (Section 3.3) and the datasets (Section 3.4) used for evaluation.

3.1. User-plane inference workflow

Figure 3.1 shows the adopted workflow for preparing, training, and deploying models into the user plane. It is divided into three parts based on where the processes are running. It begins with the pre-processing of the datasets and then moves to the feature and model selection phase which all take place in an offline ML server. Once the models are ready, they are then sent to the controller as table entries which will be loaded onto the user plane where a P4 program encoding the trained model is running. Each part of the workflow is described next.

3.1.1. Model preparation in an offline ML server

Owing to the constraints of programmable user-plane equipment described in Chapter 1, the phases of data pre-processing, model and feature selection and conversion to table entries take place offline in an ML server with adequate resources. Details about these phases are described next.

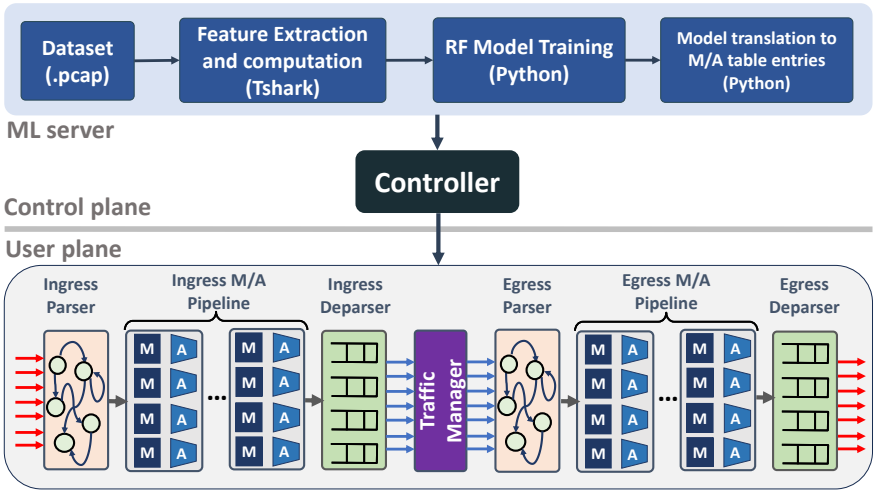


Figure 3.1: Overview of the user-plane inference workflow.

Packet-level features	Flow-level features
Packet arrival time, packet length, protocol, IP header length, time-to-live (TTL), TCP source port, TCP destination port, TCP flags (SYN, ACK, PSH, FIN, RST, ECE), TCP window size, TCP data offset, UDP source port, UDP destination port, and UDP length	Min. packet length, Max. packet length, Mean packet length, Total packet length, Packet count, Current packet length, Min. flow IAT, Max. flow IAT, Mean flow IAT, Flow duration, SYN flag count, ACK flag count, PSH flag count, FIN flag count, RST flag count, and ECE flag count

Table 3.1: List of extracted and computed features.

Data pre-processing. As in all ML tasks, the ML model preparation starts with the acquisition of the target dataset typically as packet captures in *.pcap* format. This data format is preferred because it facilitates replaying the traffic through the switch using simple tools like Tcpreplay [109]. In addition to the packet captures, the datasets often come with labelled *.csv* files or other instructions on how to label the packets or flows in the dataset. These labels are indispensable in supervised learning tasks like classification which is the focus of this thesis. Once the labelled dataset is available, Tshark [110], a terminal-oriented version of Wireshark, is used to extract packet header fields from the packets in the capture files and save them to *.csv* files as follows.

```
tshark -r $pcap_file -T fields -e ip.src -e ip.dst -e tcp.srcport ... > $csv_file
```

These header fields are directly used as features in Packet Level (PL) inference, *e.g.*, in Henna [4] or are used to compute stateful features in Flow Level (FL) inference, *e.g.*, in Flowrest [2], by aggregating them over several packets of the same flow. The packet/flow data is then cleaned and labelled using Python scripts and then saved in *.csv* files for onward feeding into the model analysis and selection pipeline. The extracted and/or computed features are presented in Table 3.1. In FL inference solutions, both the PL and FL features are available for use, with the PL ones being those of the last observed packet of the flow under consideration.

Algorithm 1: Model and feature selection process

```
Data: features, data, labels
Result: selected_features, best_hyperparamaters, final_model
best_hyperparameters = grid_search(data, labels, features);
initial_model = train_model(data, labels, features, best_hyperparameters);
sorted_features = rank_features_by_importance(initial_model);
selected_features = [ ];
accuracy_threshold = acc_th;
for feature in sorted_features do
    selected_features.append(feature);
    model = train_model(data, labels, selected_features,
     best_hyperparameters);
    accuracy = cross_validate(model, data, labels);
    if accuracy > acc_th then
        return selected_features;
    else
        continue;
final_model = train_model(data, labels, selected_features,
 best_hyperparameters);
final_accuracy = cross_validate(final_model, data, labels);
output_results(selected_features, best_hyperparameters, final_model);
```

An important remark is that, unlike previous works [40], this thesis does not employ source or destination IP and MAC addresses as features: the rationale is that such end host identifiers allow to artificially inflate the model performance in most use cases considered in the literature and this thesis. Indeed, these use cases rely on measurement data collected in limited scenarios, and a sufficiently complex model fed with IP or MAC addresses can learn during training which end-hosts are associated with the target classes (*e.g.*, belong to a specific type of device, or inject malicious traffic). Then, the inference task becomes unnaturally trivial in tests re-proposing the same end hosts, as in most use cases adopted by previous works. Yet, the model would spectacularly fail once deployed in the wild, where it would not find the same hosts it learned to recognize artificially. By excluding such identifiers, the model is constrained to complete the assigned tasks based on inherent properties of the traffic, so that it generalizes to previously unseen hosts.

Feature and model selection. The *.csv* files from the pre-processing phase are loaded and the Scikit-Learn [73] Python-based libraries are used to conduct the two-phase feature and model selection process detailed in Algorithm 1.

The process takes as input the list of features, the dataset and the labels, and outputs the selected features, the best model hyperparameters and the final model. The data and all features are first used in Line 1 to run a grid search for model hyperparameters which are the maximum depth of trees and the number of trees in the case of RF models, or only the maximum depth in the case of DT models. For each combination of hyperparameters,

a model is trained and evaluated. The best hyperparameters are those from the most accurate model obtained at this stage. With them, a model (`initial_model`) is trained using all available features. The features are then ranked in order of importance (Line 3) based on their Mean Decrease Impurity (MDI) score obtained from Scikit-Learn's `feature_importances` attribute of the tree-based models.

A desirable accuracy threshold, acc_{th}, is set depending on the score of `initial_model` and the difficulty of the use case under consideration. Then, beginning with the most important feature as the lone selected feature, a model is trained and cross-validated with a 5 or 12-fold cross-validation. The score is compared to acc_{th} which if exceeded, the selected feature is returned as the smallest set of features that enables the model to exceed acc_{th}. If acc_{th} is not attained, the next important feature is appended to the list and the steps from Line 8 to Line 13 are repeated until acc_{th} is exceeded. If acc_{th} cannot be obtained, the `initial_model` is retained. The selected features are used alongside the best hyperparameters to train the final model which is tested to validate its performance. It is then saved for later conversion into Match & Action Table (MAT) entries.

3.1.2. Model mapping to the switch M/A pipeline

The final model from the feature and model selection phase is mapped onto the switch pipeline via MAT entries that will be sent to the controller which will load them to the switch at runtime as shown in Figure 3.1. The model mapping adopted is a reproduction of the mapping proposed in Planter [40, 43]. The rationale for this choice is that, as discussed in Chapter 2, this is a state-of-the-art mapping strategy ensuring scalable multi-tree support. It has also been adopted by several previous works [10, 97, 98, 101]. The functioning and implementation of this DT/RF mapping scheme are summarized next.

The characteristic element of the mapping is that all individual Match & Action (M/A) operations involving a given feature are mapped onto a single MAT so that each feature is processed in exactly one M/A stage even if used across many levels of different trees. The rationale is that the number of features can be much smaller than the number of tree levels in complex problems. Moreover, the mappings proposed in pForest [41] and Soter [96] force each tree level to be mapped to a different M/A stage, bounding the maximum tree depth to the number of stages in a pipe which is only 12 in first generation Intel Tofino chips and 20 in the second generation. In contrast, the Planter approach decouples the levels from the M/A stages and allows the compiler to store the M/A operations of multiple features in the same stage when possible. This approach implies that all decisions based on the same feature at all nodes of all trees in an RF must be handled at once. This is illustrated by the toy example in Figure 3.2. Bold values denote the actual M/A sequence taken by a sample packet whose features are A=**3** and B=**6**. The diagram models a single tree but is generalizable to the forest case where each tree follows the same process and the final result is a majority vote of individual tree outcomes.

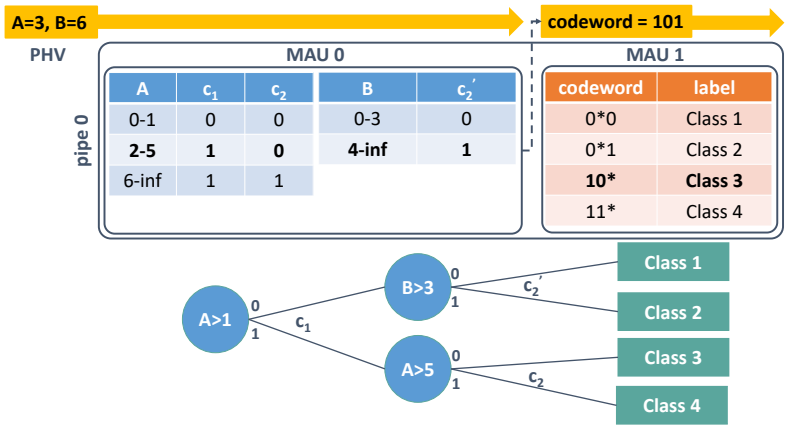


Figure 3.2: Overview of the RF model mapping to MAU stages.

First, value ranges are computed for each feature to capture all thresholds that the feature may encounter in the DT or RF. These ranges constitute the keys to be matched at inference. For instance, in Figure 3.2, the feature A is compared to thresholds 1 and 5 at nodes C1 and C2, respectively: hence, the MAT associated with A will have three value ranges, *i.e.*, less than or equal to 1, between 2 and 5, or 6 and above. Then, for each node where the feature is used, a result of true (1) or false (0) is associated with each value range above in the action stage depending on whether it is matched or not. For instance, in Figure 3.2, the first assigned code at node C1 (which checks if $A > 1$) is set to 0 for the match row where $A \in [0, 1]$, and to 1 for the two rows where A is strictly greater than one, *i.e.*, $A \in [2, 5]$ and $A \in [6, \infty)$. The second assigned code at node C2 is instead set to 1 only in the last case, where $A \in [6, \infty)$, and to 0 everywhere else. Through these M/A operations, a packet with A=3 is assigned a code 10 for feature A and this code is stored in the Packet Header Vector (PHV) metadata.

By repeating the process for each feature and concatenating the assigned codes in the PHV, a full *codeword* is generated for the packet. In the M/A stage after the feature tables, each tree in the forest has a code table with a dedicated M/A sequence, where the codeword is matched against predefined sequences that describe all possible paths from the root node to the leaf nodes of the tree. This allows matching the packet with a specific path, hence its associated label. An important remark is that not all possible codewords are feasible in each tree since each bit refers to one node, and the path to a given leaf only traverses a subset of nodes. Thus, wildcards are employed in the code tables to represent bits (*i.e.*, nodes) that are not along the path, and whose value is thus irrelevant to the current classification decision. In the case of DT models, the assignment of a class in the code table marks the end of the classification process but in the case of RFs, the final result is obtained by a majority vote of the individual tree results. This is implemented via a voting table which matches the class outputs of all the trees and assigns the final class. The entries of this table are also pre-computed offline.

3.1.3. Control plane implementation

In the solutions developed in this thesis, the ML inference process is kept entirely in the user plane. As such, the control plane is kept completely off the critical path of the inference process to minimize any latency that back-and-forth interactions between the user plane and the control plane will induce. However, the controller still plays a vital role in user-plane inference, sitting logically between the ML server and the user plane as portrayed in Figure 3.1. First, it performs the initial setup of the switch hardware by configuring ports and defining forwarding rules. Second, it loads the trained model in the form of MAT entries and injects it into the switch. The Python-based controller communicates with the user plane switches through a control plane specification, *e.g.*, the P4Runtime API [111] which is fully open-source or the Barefoot Runtime Interface (BRI) [112] provided by Intel, and which is used in this thesis since Intel Tofino Switches are the main hardware employed. Third, in use cases where the controller is expected to respond to classification results by inserting or modifying forwarding rules in the switch, the controller also listens on the interface with the switch to collect packet digests which contain summary information that the switch sends to the controller when triggered by user-specified events. Based on the information in the digests, the controller can modify table entries to change the behaviour of the switch program. In the case of FL inference solutions, the controller can also empty stateful registers to free up space to store data for newly arriving flows.

3.1.4. User plane implementation

The whole inference process takes place in the switch, *i.e.*, in the user plane as depicted at the bottom of Figure 3.1. The internal structure of the switch is based on the Tofino Native Architecture (TNA) [113] which is very similar to the Protocol Independent Switch Architecture (PISA) pipeline described in Chapter 2, differing mainly in the separation of the ingress and egress pipelines by a traffic manager. Trained DT/RF models are deployed in the switch as P4 programs which are made up of parsers, control blocks, and deparsers, with each of the ingress and egress pipelines programmed separately. The parser is programmed to extract all header fields that serve directly as PL features or which are later used to compute stateful features in FL inference as described in Section 3.1.1. The extracted header fields are stored as metadata in the PHV which runs across the switch pipeline making the metadata available to all M/A stages.

In user-plane inference use cases, a decision on the packet is desirable before a forwarding decision is assigned and since forwarding ports are assigned in the ingress pipeline, inference generally takes place in the ingress pipeline. In this pipeline, MATs are defined for the model's feature tables, code tables, and voting tables if necessary. In addition, in the case of flow-level inference, stateful registers are also declared to store and

compute stateful flow-level features using the header fields in the PHV. Any control logic required for computing features or checking conditions that trigger different actions is also implemented in the ingress control and the model tables are then applied. The nature and complexity of the control logic heavily depend on the P4 program to be implemented which in turn depends on the specific use case.

Packets exit the ingress pipeline after going through the ingress deparser where any packet digests are packed for the controller, and the parsed headers are rearranged into the packets. They are then sent to the egress pipeline after going through the traffic manager which is also a packet replication engine. In the egress pipeline, the packet is again parsed and additional processing can be applied to the packet without changing its forwarding port which is in the ingress intrinsic metadata and can no longer be modified at this stage. This is the case in Henna [4] which is presented in Chapter 5, where a second set of ML models are applied to packets in the egress pipeline for additional inference. In all other cases presented in this thesis, no additional processing is done in the egress and the packet is re-assembled and pushed onto the wire as it came from the ingress. Egress processing can be skipped altogether to save a few nanoseconds of latency if it is known in advance that no processing will be required in the egress.

3.2. Hardware testbed setup

In the initial stages of this PhD, production-grade switch hardware switches were not yet available for research. As such, early P4 development and experimentation were done on Behavioural Model version 2 (BMv2) software switches [114] connected to virtual hosts in Mininet [115]. However, development in BMv2 is not subject to all the strict resource constraints of hardware switches and so P4 programs written for BMv2 are not always feasible in hardware. Consequently, the hardware testbed shown in Figure 3.3 was assembled for development and experimentation in real-world equipment.

The hardware testbed comprises two off-the-shelf servers (Server 1 and Server 2) and three programmable switches. The servers are DELL PowerEdge servers with Intel 8-core Xeon processors at 2GHz frequency, with 48GB of RAM, and equipped with Mellanox ConnectX-5 NICs with 100 Gbps QSFP28 interfaces. The switches are the Edgecore Wedge 100BF-QS model, with Intel Tofino [23] BFN-T10-032Q chipsets and 32 100-GbE QSFP28 ports each. The links between the switches and the servers are therefore all at 100 Gbps. The switches run the Open Network Linux (ONL) operating system, and Intel's Software Development Emulator (SDE) version 9.7.0 that is used for compiling P4 programs for the TNA. The SDE is also equipped with the Intel P4 Insight tool [116] which provides a graphical user interface with a detailed view of the mapping of P4 programs onto Tofino switch hardware resources. This tool enables a thorough assessment of the consumption of switch hardware resources by compiled P4 programs and is employed in

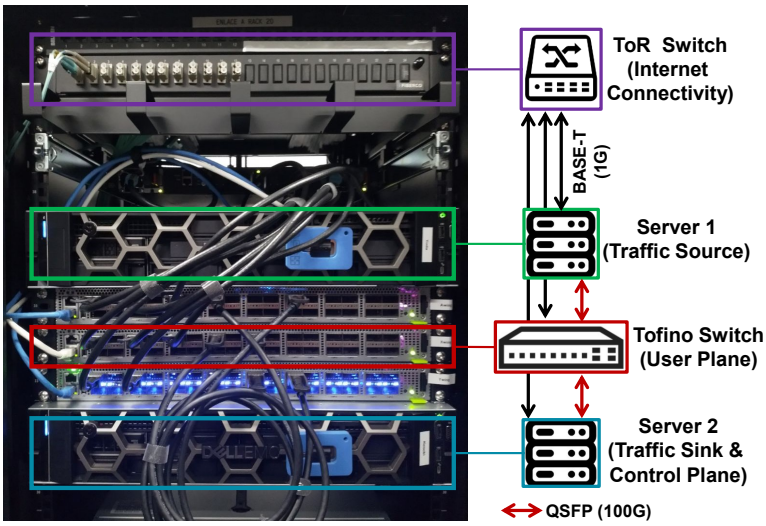


Figure 3.3: Annotated picture of the experimental testbed.

this thesis to assess the resource footprint of all deployed programs. All the switches and servers are connected to a Top-of-Rack (ToR) switch via 1000 Base-T (1 Gbps) links which provide internet connectivity and remote access.

Juxtaposing the testbed components with the user-plane inference workflow presented in Figure 3.1, the elements can be mapped as follows. The entire control plane runs in Server 2. Its 8-core processor and 48 GB of RAM provide adequate resources for the data analysis and model training phases of the workflow. It also runs the Python-based control program which prepares the encoded model for deployment in the user plane and serves to configure the switch at setup and load the model in the form of MAT entries. The Tofino switch runs the user-plane portion of the workflow, executing all the packet-processing steps that implement the ML models from the parsing phase where features are extracted, through the M/A stages implementing the models in MATs, to the deparser which packages the packet and prepares it to be forwarded.

During experiments, models are implemented in the switch as P4 programs for PL, FL, or joint inference. Server 1 serves as a traffic source which injects traffic from the test *.pcap* files into the switch using Tcpreplay [109]. Upon inference, the traffic is forwarded to Server 2, where it is captured by a Tcpdump [117] instance and saved in a *.pcap* file for onward analysis. In FL inference tasks, classification statistics are directly received in the controller via packet digests so capturing the traffic is unnecessary. The Moongen [118] traffic generator is employed to generate and send Gbps-level background traffic into the switch, to provide a more realistic traffic mix. The models do not classify background traffic and hence it does not affect classification results. It only serves to demonstrate that the user-plane inference solutions are effective even in the presence of high volumes of concurrent non-target traffic.

3.3. Evaluation metrics

The models developed in this thesis as well as the benchmarks used for comparison are evaluated using several performance metrics. These metrics are derived from the four main measures of classification tasks, *i.e.*, true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN). The metrics are explained next.

- **Accuracy** captures the amount of samples that are predicted correctly, as (TP + TN) / (TP + FP + TN + FN).

- **True Positive Rate (TPR)** measures the number of positive samples that are rightly classified as positive, *i.e.*, TP / (TP + FN). It is also known as the recall.

- **False Positive Rate (FPR)** is the fraction of negative samples incorrectly classified as positive. It is computed as FP / (FP + TN).

- **True Negative Rate (TNR)** quantifies the number of negative samples that are correctly classified as negative, *i.e.*, TN / (TN + FP).

- **False Negative Rate (FNR)** represents the portion of positive samples that are wrongly labelled negative, *i.e.*, FN / (FN + TP).

- **Precision** captures the fraction of positive predictions belonging to that class, as TP / (TP + FP). The precision depicts the ability of the classifier to not classify a negative sample as positive.

- **F1-score** is the harmonic mean of precision and recall. It is widely used to compare overall model performances in classification tasks. It is calculated as 2TP /(2TP + FP + FN).

In classification tasks with multiple classes, apart from the accuracy, all the other metrics are computed for each class, and the final value is computed in one of three ways.

- **Micro average.** It is computed by summing all the individual TP, TN, FP and FN, and computing the final metric.

- **Macro average.** It is a simple average of all individual class scores depicting the model performance in a scenario where all classes are of equal importance.

- **Weighted average.** It is an average of individual class scores weighted using the number of samples of each class. It accounts for imbalance in datasets and better represents the score of any random sample in the dataset.

In the evaluation of FL solutions, as the main goal is to classify flows, a direct comparison of FL and PL solutions would be unfair. To achieve a fair comparison, a

FL score is derived from the PL classifications of the PL solutions. This is achieved by normalizing the prediction of each packet by the length of the flow it belongs to, *i.e.*, each packet prediction is assigned a weight that is inversely proportional to its flow length. Given a packet belonging to a flow of length l, the weight of the packet prediction is given by $w = \frac{1}{l}$. This means the weights of all the PL predictions of each flow sum to one, while the weights of the predictions of all the packets in the dataset sum to the total number of flows in it.

3.4. Datasets

The solutions proposed in this thesis are evaluated using measurement data from the publicly available datasets described below.

3.4.1. Unencrypted traffic

UNSW-IoT [78] contains measurement data for 28 Internet-of-Things (IoT) and non-IoT devices, collected in a living lab emulating a smart environment. It comprises $19,871,591$ packets spread into $492,961$ flows. The models are trained on 15 days of data and tested on 1 day to distinguish 26 device types namely; Dropcam, HP Printer, Netatmo Welcome, Withings Smart Baby Monitor, Netatmo weather station, Smart Things, Amazon Echo, Samsung SmartCam, TP-Link Day Night Cloud camera, Triby Speaker, Belkin Wemo switch, TP-Link Smart plug, PIX-STAR Photo-frame, Belkin wemo motion sensor, Samsung Galaxy Tab, NEST Protect smoke alarm, Withings Smart scale, IPhone, MacBook, Withings Aura smart sleep sensor, Light Bulbs LiFX Smart Bulb, Blipcare Blood Pressure meter, iHome, Insteon Camera, Android, and Laptop.

IoT-23 [119] is a comprehensive IoT dataset including real and labelled IoT malware infections, as well as benign IoT traffic. It contains packets captured between 2018 and 2019 from infected bots and benign real IoT devices such as Philips HUE LED smart home, Amazon echo smart home personal assistant, and Somfy smart door lock in controlled environments. The IoT-23 dataset is used to design an IoT bot classification use case where 4 benign traffic classes are distinguished from 10 malicious traffic classes, generated by different bots. It is made up of $2,015,636$ packets which make up $239,182$ flows. A portion of the dataset with all classes represented is selected and then a 75-25 train-test split ratio is used in the experiments.

ToN-IoT [120] comprises benign traffic and 9 kinds of IoT cyberattacks, generated from measurements on a medium-scale experimental platform deployed using various virtual machines such as Windows, Linux, and Kali Linux operating system hosts, to manage the connections between the Cloud, Fog, and Edge network domains. The measurements were carried out with seven IoT and industrial IoT (IIoT) devices, *i.e.*, weather and Modbus sensors, a GPS tracker, a motion light, a garage door, a thermostat,

and a fridge, as well as some non-IoT devices such as a smart TV and two iPhone 7's. A 75-25 ratio is adopted for splitting the data into a train and a test set. There are 5,843,211 packets in the dataset, distributed into 240,800 flows.

UNIBS-2009 [79, 80] is a service classification dataset based on real-world traces collected on the edge router of the University of Brescia campus network, capturing traffic from 20 workstations. The traces include web traffic (HTTP/HTTPS), mail (POP3, IMAP4, SMTP), peer-to-peer applications (BitTorrent, Edonkey) and other protocols (FTP, SSH). The data comprises 3,666,016 packets and 40,338 flows. The goal is to associate each traffic flow to one of the 8 application categories. One day of traffic is used for training and another for testing.

CICIDS2017 [81] is an intrusion detection dataset based on measurements collected on a testbed at the University of New Brunswick. The testbed includes two networks: a victim network, which is a secure infrastructure with a set of computers running a daemon which implements benign behaviours; and an attack network performing 7 types of attack. The Friday dataset made up of 3,839,575 packets and 625,956 flows is used for evaluation. The goal is the binary classification of flows into benign and malicious classes. A 75-25 split is used to divide the data into train and test sets.

3.4.2. Encrypted traffic

NIMS Instant Messaging Application (IMA) [121,122] is an encrypted instant messaging application dataset. It comes as seven files in the zip format. Six of these files contain traffic from commonly used IMAs; *Discord*, Facebook *Messenger*, *Signal*, Microsoft *Teams*, *Telegram* and *WhatsApp*. The 7^{th} file contains encrypted traffic that is not from any of the IMAs, divided into 4 classes. The dataset contains 4,062,861 packets from 92,361 flows from these 10 different classes. In this thesis, only the problem of distinguishing between the 6 IMAs is considered, as in [121].

Netflow QUIC [123] is a labelled dataset of QUIC traffic. QUIC is widely used today and several works have targeted QUIC traffic classification [123,124]. The Netflow dataset has 5 different classes: *Google Hangout Chat*, *Google Hangout VoIP*, *File Transfer*, *Youtube*, and *Google Play Music*. This dataset is significantly large, with 365,000 flows and a total of 136 million packets. The distribution of flows across the 5 classes is not well-balanced with VoIP traffic being the majority. The inference task is to classify traffic into one of 5 classes.

ISCX-VPN-NonVPN-2016 [125] is used to design a Virtual Private Network (VPN) traffic classification use case. It is a popular labelled dataset made available by the Canadian Institute of Cybersecurity (CIC) at the University of New Brunswick (UNB). It comprises about 28 GB of traffic data captured using tcpdump and Wireshark. A subset of this dataset is made up of VPN data, generated using an external VPN service provider connected to the testbed using OpenVPN in UDP mode. This subset of

the dataset includes 7 classes of encrypted traffic: *Browsing*, *Email*, *Chat*, *Streaming*, *File Transfer*, *VoIP*, and *P2P*. The dataset was preprocessed from the raw *.pcap* files to keep only the VPN subset (ISCX VPN) and the packets were aggregated into $4,960$ flows.

3.4.3. Smart grid traffic

The **DNP3 intrusion detection dataset** [126] is described in [127] and analyzed in [128]. The Distributed Network Protocol 3 (DNP3) is a commonly used protocol in industrial control systems especially in smart grids, with over 75% of North American electric utilities employing it for control applications [129]. The protocol suffers from many security vulnerabilities by design and these were exploited to launch eight DNP3 attack scenarios each for a period of 4 hours from 14-19 May 2020 using opendnp3, Nmap and Scapy to generate the dataset [127]. The attacks from these scenarios include DNP3 Disable Unsolicited Messages, DNP3 Cold Restart Message, DNP3 Warm Restart Message, DNP3 Data Initialisation, DNP3 Replay, DNP3 Stop Application, DNP3 Enumerate, DNP3 Info, Man-in-the-Middle, and DNP3 Slave Discovery.

Part I

Packet-level inference

Packet-level classification is the most straightforward approach to user-plane inference because all packets are processed individually as they go through user-plane equipment like switches. As such, the vast majority of early works on user-plane inference focused on packet-level inference where each packet is classified based only on the features extracted from its headers [3, 4, 10, 38–40, 43, 93, 96].

This part of the thesis presents a practical application of packet-level inference in programmable switches to demonstrate the operation of user-plane inference at packet-level using flat classification. It then introduces a novel framework for hierarchical packet-level inference to address scalability issues in existing flat classification ML models.

4 Packet-level inference in smart grid networks

Recent cyberattacks on Smart Grid (SG) networks have shown how the evolution of power grids into complex smart systems with millions of interconnected components exposes them to attacks. The multi-event cyberattacks on Ukrainian critical industrial control system infrastructure with two disruptive events in October 2022 [130] and the hacking of India's power grid in 2021 [131] show how cyberattacks on SGs have become a growing security concern to the power industry and even national governments [132, 133].

Modern SGs are equipped with Supervisory Control and Data Acquisition (SCADA) systems which are cyber-physical systems for real-time monitoring and control of the electricity distribution network [134]. Traditionally, these systems ran on private networks but as they are increasingly being connected to the Internet to reduce capital and operating expenditure, they constitute a major loophole for attackers to exploit. The introduction of 5G and 6G which highly use Software Defined Networking (SDN) and Network Function Virtualization (NFV) that massively employ HTTP and the REST Application Programming Interface (API) will worsen the exposure of SGs to cybersecurity vulnerabilities [135].

In addition, widely used SCADA system protocols like the IEC 61850 and the Distributed Network Protocol 3 (DNP3) are not designed with native security [128]. This further exposes critical infrastructure like SGs to attacks like man-in-the-middle (MITM), distributed denial of service (DDoS), masquerading, jamming and spoofing which will lead to severe consequences in case of blackouts or data thefts on the SG network [132]. Hence, the timely detection of these attacks is key to mitigating the possible damage, the reason why it has attracted much attention in the research community [136].

Cyberattack detection techniques based on Machine Learning (ML) are widely regarded as state-of-the-art in recent surveys [137], with several works to that effect. Iqbal *et al.* [138] evaluate five different models namely Decision Tree (DT), Random Forest (RF), K-Nearest Neighbours (KNN) and Logistic Regression (LR) to classify power system disturbances. They show that the RF outperforms its counterparts. Xiong *et al.* [139] introduce a Deep Learning (DL) approach that combines information

entropy quantization, Convolutional Neural Networks (CNN) and Long Short-Term Memory (LSTM) to classify abnormal traffic in time series. Cao *et al.* [140] explore the invasion pathway of false data injection attacks on cyber-physical power systems and propose a focal-loss-lightgbm ensemble classifier to detect such attacks accurately. Siniosoglou *et al.* [141] propose an intrusion detection system named MENSA which employs a novel Autoencoder-Generative Adversarial Network (GAN) architecture for detecting operational anomalies and classifying Modbus/TCP and DNP3 cyberattacks in different SG environments. While these works show the potential of ML/DL for attack detection in SGs, they do not implement the models in the SDN architecture.

Holik *et al.* [142] propose INPS, an in-network protection system for industrial control systems. The Artificial Intelligence (AI) module based on an externally trained neural network is lodged in the ONOS controller of the SDN architecture and enables attack detection in the network. El Houda *et al.* [143] propose BoostIDS, a system leveraging an efficient boosting feature selection algorithm and a Lightweight Boosting Algorithm (LBA) to timely and effectively detect attacks on SGs in an SDN environment. The model runs in the controller and the system is evaluated in Mininet. DIDEROT [144] proposes an intrusion detection and protection system comprising a first detection layer relying on a DT classifier which recognises DNP3 cyberattacks, and a second detection layer which uses an autoencoder DNN to detect DNP3 anomalies. DIDEROT interfaces with the SDN controller via a REST API and enables attack detection and mitigation. The above works enhance SG protection against cyberattacks by implementing the models in the control plane. This induces a considerable delay in detecting any security threat due to the back-and-forth communication between the user plane where the packets transit and the control plane where ML runs. The delay is in the order of one to hundreds of milliseconds [19], which may let in the system a large volume of malicious short-lived traffic, or force the SG provider to introduce substantial latency in all communications to accommodate intrusion detection procedures.

Ndonda *et al.* [145] exploit the user plane and propose a two-level intrusion detection system for SDN-based industrial control networks. The first level runs in the switch, exploiting flow and Modbus whitelists implemented in P4 for efficient real-time monitoring. The second level is a Deep Packet Inspection (DPI) engine communicating with an SDN controller to update the whitelists of the first level. However, the solution is not ML-based and requires a DPI engine running on an external host which does not run at line rate and could entail additional costs.

Advances in user plane programmability have given rise to several proposals to embed ML models into network equipment for high-speed inference in the user plane as discussed in Chapter 2. The targeted use cases across existing works include device identification [2,4,8,38,43], intrusion detection and attack classification [2,8,38,43,93,101], service or application classification [2, 8, 101], heavy-hitter detection [44], and encrypted

traffic classification [5]. To the best of available knowledge, no work to date has tackled the case of cyberattack detection in SGs, neither has any of the existing works been evaluated on a relevant SG or SCADA system dataset.

This chapter closes the aforementioned gap by exploring for the first time how network programmability can enable SG security at ultra-low latency. The proposed approach operates at ultra-low latency of tens to hundreds of nanoseconds, hence *reducing the threat identification delay by 4–6 orders of magnitude* with respect to current standards. The main contributions of the chapter are summarized as follows.

- A workflow is proposed which brings the concept of in-switch inference to the realm of SG systems, employing the P4 programming language to translate trained switch-specific tree-based models and port them onto the programmable switch pipeline to rapidly detect and mitigate cyberattacks on SG networks via pure user-plane inference.

- The solution is implemented as P4 software and encoded into an off-the-shelf Intel Tofino switch demonstrating its feasibility in real-world equipment. The source code is made public[1] to foster reproduction and encourage its adoption.

- An experimental evaluation of the proposed solution is conducted based on measurement data from a recent DNP3 intrusion detection dataset. Results shed light on how the model can detect 6 cyberattack types with an accuracy of 99%, with only 356 nanoseconds of packet-processing latency and consuming less than 5% of each of the scarce switch resources.

These contributions advance the state-of-the-art in cyberattack detection and mitigation in SG networks.

4.1. In-switch cyberattack detection in SG networks

The cyberattack detection solution is designed as fully in-switch framework which runs entirely in the user plane, detecting and mitigating cyberattacks on SGs without any involvement of the controller in the inference process as in most previous works. The proposed workflow is described in Section 4.1.1, followed by a description of the modelling process in Section 4.1.2 and the actual in-switch implementation in Section 4.1.3.

4.1.1. Solution overview

An overview of the proposed solution is presented in Figure 4.1 alongside the workflows of closely related works like INPS [142], BoostIDS [143], DIDEROT [144] and Ndonda *et*

[1]The code is available at https://www.github.com/nds-group/smart-grid

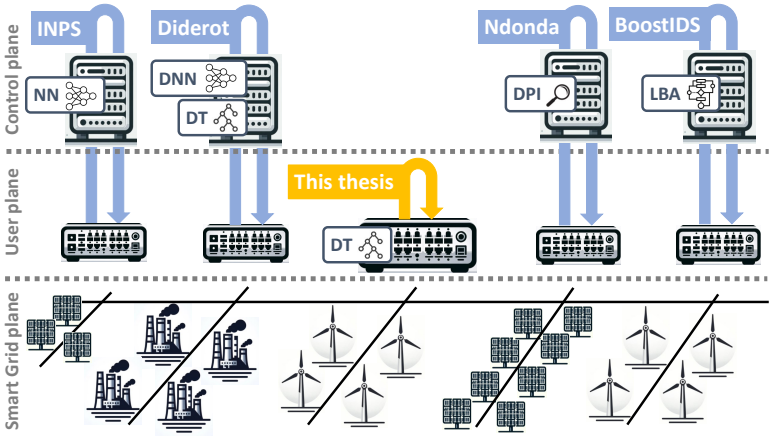


Figure 4.1: Workflows of the proposed solution and representative approaches.

al. [145] which all involve either the control plane or external hosts in their proposals. This chapter deviates from those paradigms by performing inference entirely in user-plane switches. The SDN-based SG architecture is presented as comprising three logical planes namely *(i) the smart grid plane*; which consists of the power system network and associated devices like smart meters and power distribution units, *(ii) the user plane*; which is the communication network made up of switches and routers, and *(iii) the control plane*; which lodges SDN controllers like RYU, OpenDaylight and Floodlight which implement control functions and perform configuration of the user plane [146].

The user plane lies at the centre of this architecture, playing the critical role of transmitting sensitive data that could include consumption data, grid status, and control commands. This makes it a juicy target for many cyberattacks against the SG network. Most existing approaches to detect and mitigate cyberattacks run in the control plane or in external systems connected to the control plane as illustrated in Figure 4.1, which introduces significant additional delay required for the user plane to communicate with the higher plane either to transmit features for real-time classification or to receive instructions on post-classification configurations. The proposed solution allows for eliminating this latency by embedding ML models into the user-plane switches, so that attack detection occurs simultaneously as data transmission. This enables line rate traffic classification with extremely low latency and high throughput, and reduction of the reaction time to attacks by applying mitigation techniques immediately after detection.

4.1.2. Model preparation

Prior works have demonstrated the potential of DT and RF models for attack detection and classification in SG systems [128, 144, 147]. As these models are also the most adaptable to switch constraints as discussed in Chapter 2, they are also adopted for this solution. The model preparation phase also happens entirely offline in an ML server.

The model preparation phase begins with the extraction of 13 features from the packet headers. They include packet length, source port, destination port, protocol, TCP flags (SYN, ACK, FIN, PSH, RST), TCP header length, TCP window size, UDP length, and time-to-live (TTL). Once extracted, the packets are then labelled using the label information provided by the dataset and then saved in CSV files. Using only header fields ensures that data privacy is maintained as payloads are never inspected.

The feature and model selection follows the approach described in Chapter 3 and employed by other works included in this thesis [2, 4, 5, 8]. For the grid search of model hyperparameters, the number of trees in the RF is limited to the range $[1, 7]$ as very large RFs will be infeasible for in-switch implementation. Unlike in other works, the maximum depth of the DT/RF models is not limited and the trees are allowed to grow to full size, maximizing their learning abilities. In the end, the best model is selected based on a trade-off between accuracy and complexity.

Once the final model and its features are selected, the model is trained and then converted to Match & Action Table (MAT) entries that will be loaded onto the switch. This conversion employs the mapping scheme described in Chapter 3.

4.1.3. User-plane model implementation

The per-packet in-switch solution is designed for the Tofino Native Architecture (TNA) shown in Chapter 3. The different components of the in-switch solution are implemented into this architecture as follows.

Parsing. Packets arriving the switch are processed by the parser which extracts their header fields and saves them in metadata. Amongst these header fields are the features required for the inference task, which are carried by the Packet Header Vector (PHV) to the Match & Action (M/A) pipeline where they will be used for inference.

Traffic filtering. Upon entry into the ingress pipeline, a first MAT known as the traffic filter is applied. The purpose of this table is to match the source IP, destination IP, source port, destination port and protocol of the packet to determine if it is part of the traffic targeted for inference and which should be processed, or just background traffic which should be forwarded normally. This avoids unnecessary processing of all packets, improving efficiency.

Packet classification. If the packet is part of the target traffic, its features (header fields) are retrieved from the metadata in the PHV and then the feature tables are applied to assign codes which when concatenated will generate a code word. The tree code tables are then applied to match the generated code word to the corresponding leaf node which has an assigned class. In the case of RFs the final class is obtained as a majority vote of the tree classes.

Attack mitigation. Once the packet is assigned a class, a class-based action can be performed. In this chapter, this is demonstrated by an attack mitigation MAT which

matches the class of the packet and forwards it normally if it is benign or drops it if classified as malicious. More advanced post-classification functions can be implemented depending on the needs of the SG system.

Deparsing. When the packet is classified and a forwarding action is assigned, the packet is packaged in the deparser and then sent to the egress pipeline through the traffic manager. In this chapter, no additional processing is done in the egress and the packet is directly forwarded or dropped depending on the action assigned by the mitigation table. Nonetheless, more processing could take place in the egress pipeline, *e.g.*, applying other models for additional levels of classification as will be done in Chapter 5.

4.2. Experimental evaluation

The solution is implemented in an Intel Tofino switch as a P4 program. Experiments are then conducted with measurement data in the hardware testbed presented in Chapter 3 to validate the proposed solution.

4.2.1. Use case: DNP3 attack classification

The proposed solution is evaluated using the recent DNP3 intrusion detection dataset [126] which has 8 attack scenarios and is described in Chapter 3. Apart from the slave discovery scenario, the dataset comes with a folder for each attack scenario, as well as three other folders for the man-in-the-middle DoS, DNP3 enumerate and DNP3 info scenarios, which are not described in [127]. The first six attack scenarios alongside the normal traffic they contain are therefore the seven classes targeted in this evaluation. The composition of the dataset in terms of the number of packets of each of the classes reveals an imbalance in the distribution, with the Replay Message class having the least number of packets. This imbalance affects the ability of the model to learn to distinguish all the classes, affecting the classification results as shown in Section 4.3.1.

Each scenario folder has labelled CSV files from which the corresponding labels for every flow are extracted. Then using Scapy, all the *.pcap* files in the respective *Total PCAP Files* folders are split into train and test *.pcap* files, using the $75 - 25$ split ratio. The features listed in Section 4.1.2 are then extracted from the *.pcap* files and saved to labelled CSV files using Tshark and Python. The model and feature selection algorithm is then executed for DT and RF model structures beginning with the full set of available features. The final selected model is the best DT which uses 4 features namely `source port`, `destination port`, `TCP data offset`, and `packet length`, and has similar scores with the best RF as will be shown in Table 4.1. It is selected for in-switch implementation due to its simpler nature which implies less switch resource usage.

4.2.2. Evaluation metrics and benchmarks

The classification performance is evaluated using five of the performance metrics presented in Chapter 3. They include the True Positive Rate (TPR), True Negative Rate (TNR), False Positive Rate (FPR), False Negative Rate (FNR), and the F1-score.

The performance of the in-switch solution is compared to that of a fully-grown offline RF model, running in an ML server which is oblivious to the constraints of the switch environment. This model is the best RF model obtained from the model and feature selection process and it has 3 trees and 4 features namely `source port`, `destination port`, `TCP window size`, and `packet length`. It is representative of prior work for cyberattack detection in SDN-based SG where the models run in external servers or the control plane, *e.g.*, INPS [142], BoostIDS [143] and DIDEROT [144].

To demonstrate the ability of the proposed solution to enable inference with a packet-processing latency in the sub-microsecond range, two additional control plane benchmarks are introduced. In the first case, the switch acts as a feature extraction engine, extracting the required features and sending them as a packet digest to the controller. There, they are used to execute the trained model and classify the packet. This is similar to the approach employed by previous works like Flowlens [32], Marina [35] and Peregrine [33, 34]. In the second scenario, the switch sends the packet to its CPU which is in the control plane. The CPU then extracts the features from the packet using Scapy and then infers on the packet using the trained model. This method is similar to the one employed in Soter [96] where packets are sent to the CPU for fine-grained classification with a complex DNN.

4.3. Results and discussion

The SG cyberattack detection solution is evaluated in terms of its classification performance (Section 4.3.1), consumption of scarce switch resources (Section 4.3.2), and packet processing latency (Section 4.3.3).

4.3.1. Classification accuracy

The per-class classification performance of the in-switch solution and the offline benchmark in terms of the metrics presented in Section 4.2.2 are shown in Table 4.1. The models are evaluated based on their ability to correctly identify 7 classes which are benign traffic and 6 cyberattacks. Two experiments are conducted. In the first experiment, the model is evaluated on classifying all the attacks and normal traffic. Overall, the in-switch model correctly classifies 99.96% of all packets. Breaking this down into individual class scores, the model achieves F1-scores of between $90.90\% - 100\%$, TPRs of between $85.36\% - 100\%$, and FNRs of between $99.95\% - 100\%$. In fact, the only class where the model achieves less than 99% of any of the above scores is the Replay Message attack,

	F1-Score		TPR		FPR		TNR		FNR	
	Offline	**In-Switch**	**Offline**	**In-Switch**	**Offline**	**In-Switch**	**Offline**	**In-Switch**	**Offline**	**In-Switch**
Normal	99.936	99.934	99.984	99.981	0.050	0.050	99.950	99.950	0.016	0.019
Disable Unsolicited	100.000	100.000	100.000	100.000	0.000	0.000	100.000	100.000	0.000	0.000
Cold Restart	100.000	100.000	100.000	100.000	0.000	0.000	100.000	100.000	0.000	0.000
Warm Restart	100.000	100.000	100.000	100.000	0.000	0.000	100.000	100.000	0.000	0.000
Data Initialization	100.000	100.000	100.000	100.000	0.000	0.000	100.000	100.000	0.000	0.000
Replay	91.162	90.900	85.489	85.360	0.005	0.006	99.995	99.994	14.511	14.640
Stop Application	100.000	100.000	100.000	100.000	0.000	0.000	100.000	100.000	0.000	0.000
Macro average	**98.728**	**98.690**	**97.925**	**97.906**	**0.008**	**0.008**	**99.992**	**99.992**	**2.075**	**2.094**
Weighted average	**99.959**	**99.958**	**99.961**	**99.959**	**0.015**	**0.015**	**99.985**	**99.985**	**0.039**	**0.041**

Table 4.1: Classification results of the in-switch solution and the offline benchmark model.

where it achieves F1 and TPR of 90.90% and 85.36% respectively. This is attributed to the small number of samples of this attack in the dataset, which prevents the model from learning to predict it accurately. This effect is also perceivable in the FPR and FNR, where the model achieves 0.006% and 14.41% respectively, suggesting that larger training datasets or techniques to handle training data imbalances could be used to properly detect Replay Message attacks.

For all the other attack types, the FPR and FNR are all 0% as shown in Table 4.1. These results shed light on how the model perfectly detects 5 of the 6 attacks, and also correctly identifies 85% of the more challenging Replay Message attack. Comparing these scores to those of the offline model also presented in Table 4.1, it is evident that the in-switch model scores align closely, confirming the efficiency of the model encoding in the user plane, despite approximations like the rounding up of thresholds at tree nodes.

To evaluate the mitigation function of the solution, a second experiment is performed in which all packets belonging to any of the attack classes are automatically dropped by the attack mitigation MAT, while only benign traffic is forwarded. At the receiver, only 0.11% of the packets wrongly arrive there as benign; these packets all belong to the Replay Message class with the lowest accuracy as discussed previously. This demonstrates that the mitigation module is efficient in its role of dropping malicious packets immediately after they are classified as such, without any involvement of the controller. While this serves as a simple proof of concept, any other mitigation strategy beyond packet discarding could be implemented depending on the needs of the SG system, *e.g.*, forwarding different classes of malicious traffic to honeypots for further analysis and design of adequate response policies.

4.3.2. Switch resource usage

The switch is not designed to run ML models, rather it is optimized for packet processing and forwarding. As such, deploying ML models in the switch must leave sufficient resources for other normal switch functions. The proposed in-switch inference solution's consumption of switch resources is evaluated using P4 Insight [116]. The resources assessed include Static Random-Access Memory (SRAM) used for exact match

Resource	Usage
SRAM	1.0%
TCAM	2.4%
Exact Match Input Xbar	0.6%
Ternary Match Input Xbar	3.7%
VLIW Instruction	1.8%

Table 4.2: Usage of key switch resources.

	Feature extraction & inference in control plane	Feature extraction in user plane & inference in control plane	Feature extraction & inference in user plane (This work)
User-plane processing	335 ns	341 ns	356 ns
Control plane processing	1.797 ms	0.483 ms	0
Inter-plane communication	4.189 ms	1.535 ms	0
Total	**5.986 ms**	**2.018 ms**	**356 ns**

Table 4.3: Delay induced by the different inference approaches.

tables, Ternary Content-Addressable Memory (TCAM) used for ternary and range match tables, the crossbars (Xbars) used for storing the table match keys, and the Very Long Instruction Words (VLIW) which carry mathematical operations.

As shown in Table 4.2, the model consumes less than 5% of each key switch resource. To better appreciate this resource consumption, when deployed concurrently with the standard P4 program for L2/L3 forwarding known as `switch.p4`, the cyberattack detection solution only brings an additional 7 MATs with a total of 227 table entries, increasing consumption of SRAM by only 1.44% and TCAM by 4.37% of what the switching program already needs. Another important aspect is the number of M/A stages required for the program, which is 4 out of 12. One stage is used for the traffic filter, another for the DT model feature tables, another for the code table which assigns the classes, and the last stage for the attack mitigation table. This leaves many stages for more complex mitigation or post-classification procedures to be implemented based on the needs of the SG network. Overall this modest consumption of switch resources makes the solution memory efficient, allowing it to cohabit with multiple switch applications.

4.3.3. Inference latency

Finally, the key advantage of the proposed system is that inference on packets is achieved at line rate, *i.e.*, at the same speed at which the packets transit through the programmable switch. This capability is demonstrated via measurements that are summarized in Table 4.3. The table reports the latency induced by the proposed solution as the sole packet processing delay in the switch, at 356 ns: this is the time needed to both classify and forward or drop a single data packet. This is compared to two approaches: (*i*) where feature extraction and inference take place in the control plane, with the user plane sending the entire packet to its CPU via the dedicated 10 Gbps PCIe interface; (*ii*) where the user plane extracts features and sends them to the remote controller running

on Server #2 for inference. The user plane and the control plane processing time are estimated respectively in nanoseconds and milliseconds. The results, shown in Table 4.3, are consistent with prior work on similar measurements [19] and highlight that (*i*) the proposed solution is more than $6,000$ times faster than solutions using user-plane feature extraction and control-plane inference, and (*ii*) more than $17,000$ times faster than those doing both in the control plane. This confirms its superiority in terms of attack detection and response time, and sets a new standard.

4.4. Summary

Cyberattacks on SGs have become a growing concern in recent years, triggering a strong research interest in the rapid detection and mitigation of such attacks. This chapter leverages ML inference in programmable switches to enable rapid detection and mitigation of cyberattacks on SDN-based SGs. A DT model is embedded into the switch and evaluated on a DNP3 attack detection and classification use case where it achieves 99% accuracy while consuming less than 5% of each key switch resource. This is achieved while keeping packet-processing latency in the sub-microsecond range. This sets forth a new state-of-the-art in the rapid detection of cyberattacks on SGs and showcases a practical application of user-plane inference.

Future work could consider evaluating the solution on attacks against other SG protocols and integrating the solution in a testbed with the complete SG system architecture. In addition, exploring other and potentially more complex per-packet inference use cases is another research direction that will continue to be relevant as more areas of application of user-plane inference emerge. Lastly, handling scalability issues with packet-level inference solutions is very important. In complex use cases, model size could grow to levels that render them not feasible for deployment in switches. As such, exploring new techniques to embed such models in the user plane is an interesting prospect. Chapter 5 makes a step in that direction with the proposal of a hierarchical packet-level inference framework to facilitate the in-switch deployment of complex inference models.

5 Henna: Hierarchical packet-level inference

Prior works for inference at packet level in user-plane switches have shown that the resource usage of Machine Learning (ML) models surges with model complexity [39, 40]. As the complexity, *e.g.*, the number of classes in the use case increases, the model size also often grows until it hits a performance ceiling determined by the amount of available resources in the switch. In these cases, a more complex model that could have yielded better accuracy is just not feasible for in-switch operation. Also, although the next generations of programmable switches will arguably have more resources and capabilities, some constraints like the maximum number of bits that can fit in a unit of Ternary Content-Addressable Memory (TCAM), will most likely persist and require strategies to accommodate them. Ultimately, this leads to the conclusion that *there is still a large margin for improvement in ML-based packet-level solutions for addressing relatively complex inference tasks within programmable switches.*

All existing works for packet-level inference adopt a monolithic approach to packet classification, *i.e.*, for each task and no matter how complex it is, a single Decision Tree (DT) or Random Forest (RF) model is trained and deployed in the switch. By addressing inference tasks via a single DT or RF, existing tree-based solutions do not scale well with the complexity of the inference problem as described above and their performance is restrained by the in-switch environment. In this context, hierarchical inference paradigms come to the rescue by splitting the main inference task into multiple simpler ones that are themselves easier to handle. Then, smaller classifiers can be trained to solve the sub-tasks, collectively yielding better accuracy while fitting within the limited switch resources.

This chapter builds on this concept and makes a step forward towards addressing the complexity-accuracy trade-off in existing packet-level solutions by proposing a framework for hierarchical in-switch machine learning, *i.e.*, `Henna` [4]. To the best of available knowledge, `Henna` is the first model that does not rely on a single-stage model architecture but implements a hierarchical multi-stage operation for in-switch inference. The main concept behind this solution is that a difficult classification task can sometimes be split into a sequence of easier ones, each of which can then be tackled using simpler, resource-

efficient, and performance-improving tree-based classifiers. Therefore, `Henna` builds on the concept of hierarchical or multi-stage classification, which is known to alleviate imbalances in data and simplify classification tasks by exploiting the relationships between classes in earlier stages [148]. By proposing `Henna`, this chapter makes the following contributions.

- An in-switch implementation of a hierarchical two-stage classifier is proposed which breaks down difficult classification tasks into simpler cascaded ones that are themselves easily dealt with by smaller classifiers that are adapted to the constraints of programmable switch environments.

- Both the ingress and egress processing pipelines are exploited to logically separate the two stages of `Henna`, with the first stage running in the ingress and the second stage running in the egress. This establishes `Henna` as the first in-switch inference model to exploit both parts of the pipeline, achieving a more efficient resource allocation as different models share Match & Action (M/A) stages in the ingress and egress pipelines.

- `Henna` is implemented alongside a one-stage benchmark classifier in an off-the-shelf Intel Tofino programmable switch using the P4 language. The source code is made publicly[1] available to promote research in the area of in-network inference.

- Experiments are conducted on the hardware testbed described in Chapter 3 and results demonstrate how `Henna` improves classification performance with respect to the benchmark by a relative gain of up to 21% for a device identification task with 21 classes, which is a much more complex use case than those considered in prior works on per-packet inference that targeted a maximum of 8 classes.

5.1. Henna multi-stage tree model

The most commonly used approach to classification problems is flat classification in which a monolithic global classifier is trained to identify all target classes in a single stage. While this one-stage strategy works well for problems with just a few classes or with many classes that are naively told apart, it becomes less practical in tasks where the number of classes is large, or where there is an imbalance in the representation of classes, such that the differences between them become more nuanced [149]. In such cases, monolithic models tend to become extremely complex to meet the desired accuracy, rendering them less suitable for in-switch deployment.

Fortunately, most real-world classification problems are intrinsically cast as hierarchical classification tasks in which the classes to be predicted are organized into a recognizable class hierarchy that is typically a tree or a directed acyclic graph (DAG) [150].

[1]The python and P4 code is available at https://github.com/nds-group/Henna.

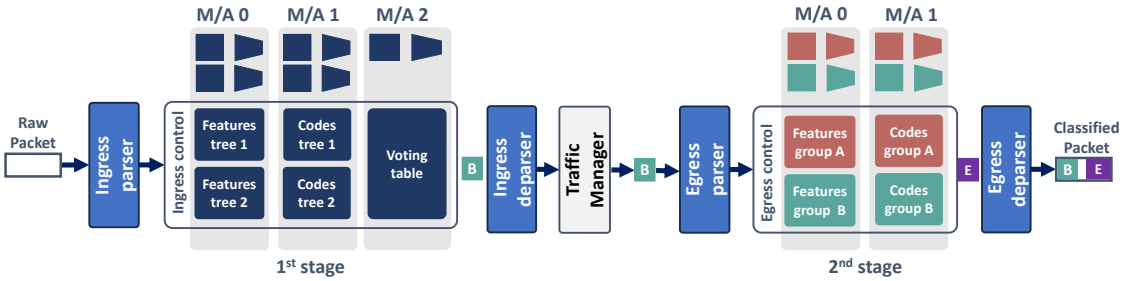


Figure 5.1: Overview of the Henna two-stage architecture mapped onto the TNA pipeline.

Hence, to address the accuracy-complexity trade-off with monolithic classifiers, the inherent hierarchical relationships between one or more classes in the problem can be exploited to simplify the classification task, by identifying class groups in earlier stages. Then in later stages, more fine-grained classification is performed on the class groups to identify other sub-class groups and later the final classes.

The concept described above forms the basis for hierarchical classification, where the task is broken down into multiple stages in a hierarchy tree or directed acyclic graph, where easier distinctions are made at higher stages and the final stage identifies the actual class of the sample. The multi-stage approach also helps to mitigate the problem of imbalance in datasets, by training local classifiers for groups of classes where each class is more likely to be better represented than in the whole dataset. Henna builds on this concept to realize hierarchical inference in switches and enable large classification tasks with inherent hierarchical relationships between the classes to be solved with simpler switch-friendly tree-based models.

5.1.1. In-switch hierarchical inference design

An overview of the proposed Henna workflow is presented in Figure 5.1. The solution is designed as a two-stage sequence of tree-based classifiers, which exploits both the ingress and egress packet processing pipelines of the Tofino Native Architecture (TNA) to classify packets at high speed and with low latency. It is made up of two main parts: (*i*) the *first stage* which is implemented in the ingress processing pipeline to perform a first level of packet classification and assign packets to groups; and (*ii*) the *second stage* implemented in the egress pipeline which performs a second level of classification and assigns a final class to the packets. The full workflow of the system and the roles of the individual components are described next.

Ingress parser. This serves as the feature extraction module of the switch. Packets arriving at the switch go through the ingress parser which extracts header information from the packets and stores them in metadata in the Packet Header Vector (PHV) as described in Chapter 3. Such header information includes data like packet length, transport protocol ports, and TCP flags, all of which serve as features for the subsequent ML-based inference steps.

Ingress control. Upon exiting the parser, the packet arrives at the ingress control which is at the start of the M/A pipeline. Once the packet arrives here with the PHV carrying the features as metadata, the first stage model in the `Henna` cascade is executed so that a class group is assigned to the packet by the first stage of the hierarchical classifier. In this thesis, this first-stage model is implemented as an RF model. The class group information is stored in a header field so that when the packet arrives at the egress, this information will also be available for use. The packet is then reassembled at the ingress deparser and sent to the egress via the traffic manager as shown in Figure 5.1.

Egress parser. As the packet arrives at the egress pipeline, it is once more parsed just like in the ingress by the egress parser, and all the necessary header information including the classification result from the previous stage is extracted and stored in the PHV. The packet then moves into the egress control.

Egress control. Here, the class group assigned by the first-stage model is checked and, based on its value, a specific second-stage model is selected for further classification of the packet. Generally, the tasks of these second-stage models are much simpler than those of the first stage. Therefore, in the current `Henna` implementation, all second-stage models are implemented as DTs[2]. The appropriate DT model is then run using the features that were extracted by the egress parser, and the final class information is obtained and stored in a header field for downstream accuracy analysis. The processed packet is reassembled at the egress deparser and forwarded to the desired switch port.

The RF and DT models are mapped to the switch's ingress and egress M/A pipelines using the mapping scheme described in Chapter 3. As shown in Figure 5.1, the feature tables of the different trees of the RF used in stage 1 are mapped unto M/A stage 0, indicating how they can share stages since they run parallel to each other. The same applies to the code tables of these RF trees which are shown to be mapped to M/A stage 1. Once the feature tables are applied for a given packet and resulting code words are generated, the code tables are then applied to assign per-tree classes to the packet. In the next M/A stage, *i.e.*, M/A stage 2 as illustrated, a voting table matches the individual tree classes and assigns a final class by majority vote. The tree code tables also output a certainty in addition to the classification result. This quantifies the confidence of the classification. In case there is no majority from the voting table or there is a tie, the certainty values are used to break the tie. In case they also cannot break the tie, the result of the tree with the highest individual accuracy is retained.

It is noteworthy that the ingress and egress pipelines are both exploited to optimize model placement and resource sharing. The RF and DT models could be both implemented in the ingress pipeline, especially if the outcomes of the second stage also had an impact on the forwarding decision which must be made in the ingress. However, this

[2]Extending the implementation of `Henna` to support RF models in the second stage is straightforward but was not required for the use case targeted in the performance evaluation presented in Section 5.2.1.

approach forces the trees in the two stages to line up in series in the ingress processing, contending for the same M/A resources, thus limiting the number of models that could co-exist within the same M/A stage. Allocating the models of the first and the second stages to the ingress and egress pipelines respectively, mitigates the problem and allows trees in both stages to share the resources of the M/A units. This is shown in Figure 5.1 where the feature tables of the second stage DTs are also mapped to M/A stage 0 like those of the first stage RF, and all the code tables are mapped to M/A stage 1, irrespective of what stage they belong. This leads to a more efficient use of the scarce switch resources.

5.1.2. Offline model preparation

Due to the limited capabilities of the switch, all ML model preparation steps are executed offline in an ML server as expatiated in Chapter 3. These include feature extraction, model training and validation, as well as the generation of all feature and code tables that allow implementing the tree-based models in the TNA pipeline as described in Section 5.1.1 above. As `Henna` is a packet-level inference solution, only packet-level features are extracted and used to run the model preparation pipeline. The features considered include TCP flags (ACK, SYN, PUSH, ECE, RESET, FIN), TCP/UDP source and destination ports, and the packet length.

The organization of the switch TCAM makes it able to support only a limited size in bits of the sequence of feature-level and tree-level codes used to map trees to the switch pipeline. This inherently limits the number of leaves that can be supported by the hardware. Thus, in `Henna`, pruning is used to limit the number of leaf nodes in trees which in turn imposes a limit on the number of nodes and hence the size of the code word. This is achieved in Python by setting the maximum number of leaf nodes using the `max_leaf_nodes` parameter of the tree models in Scikit-Learn. This ensures that the trained models fit within the switch constraints.

5.2. Experimental evaluation

`Henna` is implemented as a P4 program which can be compiled and executed on any Tofino programmable switch. Experiments are conducted in the real-world experimental testbed described in Chapter 3 using a device classification use case of much higher complexity than those considered in prior works evaluating user-plane ML models.

5.2.1. Inference use case

As hierarchical inference tends to excel in use cases with a large number of related classes, a much more complex use case than those typically considered in the related literature is considered to evaluate `Henna` and demonstrate the benefits of hierarchical

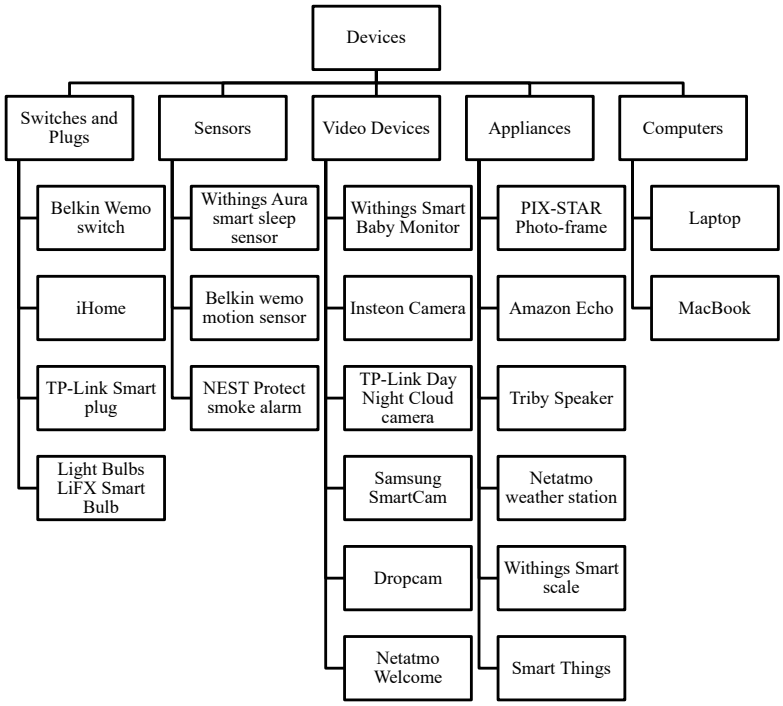


Figure 5.2: Hierarchical relationship between the target classes.

in-switch inference. To that end, the publicly available UNSW IoT traces [78] described in Chapter 3 are used. An inference task is designed where the objective is to classify packets going through the switch into one of 21 possible classes, representing the devices sending the traffic. It is important to remark that the largest classification task tackled in works before `Henna` involved a maximum of 8 classes. By targeting up to 21 classes, `Henna` makes a step towards tackling more complex use cases.

Based on domain knowledge and previous work on the same dataset [38], the 21 devices under consideration are split into 5 groups namely *Switches and Plugs* (Belkin Wemo switch, iHome, TP-Link Smart plug, and Light Bulbs LiFX Smart Bulb); *Sensors* (Withings Aura smart sleep sensor, Belkin wemo motion sensor, and NEST Protect smoke alarm); *Video Devices* (Withings Smart Baby Monitor, Insteon Camera, TP-Link Day Night Cloud camera, Samsung SmartCam, Dropcam, and Netatmo Welcome); *Appliances* (PIX-STAR Photo-frame, Amazon Echo, Triby Speaker, Netatmo weather station, Withings Smart scale, and Smart Things); and *Computers* (Laptop, and MacBook). Figure 5.2 shows the established hierarchy between the devices and their class groups.

The first stage of the proposed hierarchical model aims at identifying the group of devices to which a packet belongs using an RF model. In the second stage, 5 dedicated DT models return the actual class (device), given the group information assigned in the previous stage. For example, if a packet emanates from the Insteon Camera, the first stage RF will first identify the packet as belonging to the Video Devices group. Then in the second stage, the specific DT model trained only on the Video Devices will tell what

specific video device the packet comes from, *i.e.*, Insteon Camera in this case. Based on the model hyperparameter tuning procedure described in Chapter 3, an RF model with 3 trees of maximum depth of 10 is used in the first stage, and DT models are used in the second stage with variable maximum depth between 4 and 10, depending on the group. The models are trained using 15 days of traffic data and tested using 1 day.

5.2.2. Benchmark and metrics

`Henna` is compared to a state-of-the-art single-stage flat classification model, *i.e.*, a monolithic RF which also employs the same implementation of the Planter [40] RF mapping. This benchmark aims at classifying each of the 21 target devices at once, without any reference to possible existing hierarchical relationships between the classes. After running the same model and feature selection process as in `Henna`, the benchmark is configured with 3 trees of a maximum depth of 10. Increasing the number of trees in the RF did not result in any significant performance gain.

This model is considered to be representative of most of the previous solutions for per-packet in-switch inference, which all use a single-stage approach and so experience the same problems that `Henna` seeks to address. The single-stage benchmark and `Henna` are deployed using P4 in the Tofino switches in the hardware testbed described in Chapter 3. The quality of the classification results obtained from both `Henna` and the benchmark is evaluated using three standard performance metrics, *i.e.*, precision, recall and F1-score which are described in Chapter 3.

5.3. Results and discussion

`Henna` and the one-stage benchmark are evaluated in terms of classification accuracy as expressed by the metrics in Section 5.2.2, and also in terms of their consumption of the scarce switch resources.

5.3.1. Classification accuracy

The classification performance of `Henna` and the single-stage benchmark classifier in terms of precision, recall and F1-score are summarized in Table 5.1 and Table 5.2. `Henna` consistently outperforms the single-stage approach across all metrics, with relative gains of over 7% in terms of macro precision, 27% in terms of macro recall, and 21% in terms of macro F1-score. In terms of weighted scores, `Henna` achieves relative gains of 5% in precision, 8% in recall, and 8% in terms of F1-score.

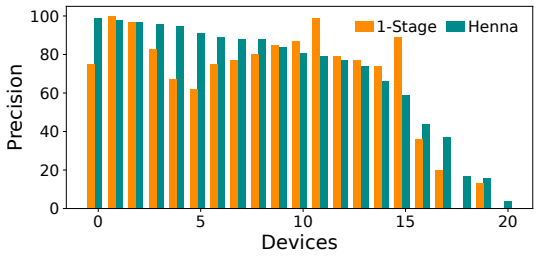
A detailed breakdown of the classification performance of the models in detecting each of the 21 devices is shown in Figures 5.3 (a), (b), and (c). The figures show how the improved accuracy of `Henna` is consistent across the target classes when considering any of

Metric	1-Stage	Henna		
		Value	Gain	
			Absolute	Relative
Precision	65.38%	70.50%	5.12%	7.83%
Recall	55.50%	70.95%	15.45%	27.84%
F1-score	55.54%	67.50%	11.95%	21.52%

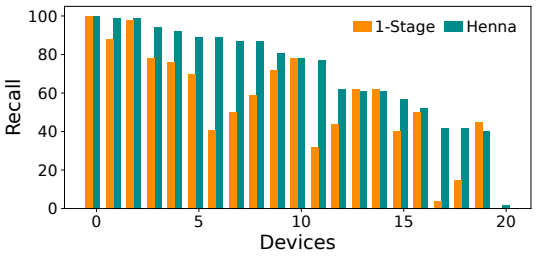
Table 5.1: Macro scores of the three considered metrics for the single-stage benchmark and `Henna`, alongside the absolute and relative gains of `Henna`.

Metric	1-Stage	Henna		
		Value	Gain	
			Absolute	Relative
Precision	84.50%	89.07%	4.57%	5.41%
Recall	77.25%	83.91%	6.66%	8.62%
F1-score	78.95%	85.52%	6.57%	8.32%

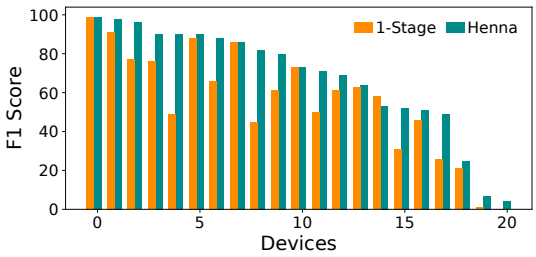
Table 5.2: Weighted scores of the three considered metrics for the single-stage benchmark and `Henna`, with absolute and relative gains of `Henna`.



(a) Precision



(b) Recall



(c) F1-score

Figure 5.3: Classification performance of `Henna` and the benchmark.

the metrics. For every class, the precision of `Henna` typically aligns with or is better than that of the single-stage benchmark, with only a few exceptions as shown in Figure 5.3 (a). `Henna` particularly stands out in terms of improving the recall, *i.e.*, in reducing the number of false negatives compared to the benchmark. The F1-score which combines precision and recall also reflects the trends above. `Henna` improves this compound metric in all classes with just one exception, where it still achieves very close performance to that of the one-stage model. Overall, these scores confirm the superiority of `Henna` over the one-stage benchmark in terms of classification performance.

5.3.2. Resource Usage

The main function of the switch is to forward traffic. As such, it is important to assess the impact of deploying additional functions in the switch and to find out if and to what extent the enhanced accuracy of `Henna` comes at an additional cost in terms of

Resource	1-Stage	Henna
Overall (w.r.t. total)	5.10%	8.50%
Overall (w.r.t. switch.p4)	13.42%	22.27%
Match-Action stages	8	10
Latency at ingress	35.42%	43.40%
Latency at egress	59.15%	62.68%
Power consumption at ingress	20.73%	15.56%
Power consumption at egress	–	42.29%

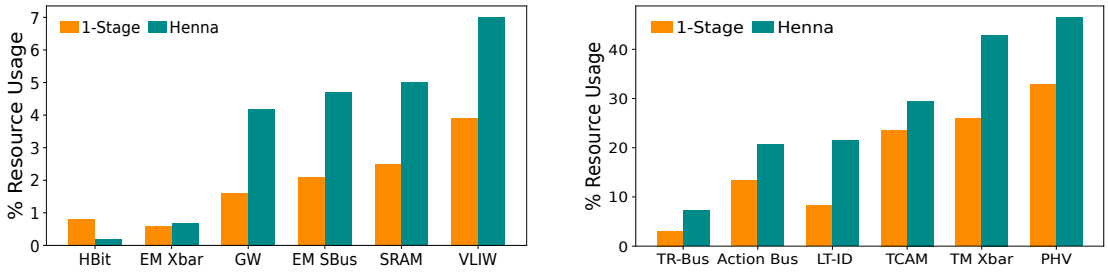
Table 5.3: Summary of the resource usage of the models. Power consumption and latency are expressed in % of those of switch.p4.

increased consumption of switch resources, packet processing latency, or estimated power consumption. The Intel P4 Insight tool [116] described in Chapter 3 is exploited to gather these statistics which are summarized in Table 5.3.

Averaging over all the considered resources, `Henna` consumes just an added 3.40% of the total available resources compared to the single-stage benchmark, for an overall 8.50% utilization of the hardware capacity. When compared to the standard P4 program for core L2/L3 switching, popularly known as `switch.p4`, `Henna` consumes just about 22% of what the baseline program requires. These figures indicate that `Henna`'s consumption of scarce switch resources is minimal and that it leaves adequate room in the switch to coexist with other legacy switching functions; this is especially true when considering that `Henna` would share M/A stages with legacy switching functions upon compilation and that it would reuse constructs (*e.g.*, header information) already created by legacy functions within limited resources like PHV containers.

The number of M/A stages used by a P4 program indicates how long the sequence of instructions that need to be executed in series is. The one-stage model requires 8 M/A stages to implement one RF model, whereas `Henna` only requires 2 more M/A stages to implement both a similarly sized RF in its first stage, plus 5 additional DT models in its second stage. `Henna` achieves this by pioneering the use of both the ingress and the egress pipeline for inference. This allows trees of different stages to coexist in the same M/A stages and thus, leads to a more efficient use of switch resources.

Concerning latency, Table 5.3 shows that the inference latency of `Henna` is much smaller than the packet processing delay of legacy forwarding functions via `switch.p4`; specifically, latency is 43.40% and 62.68% of that of `switch.p4` in ingress and egress, respectively. These figures are slightly higher than those for the one-stage benchmark, but well within the limit for line-rate operation as classification occurs much faster than packet forwarding. In any case, all compiled programs run at line rate and the comparison above sheds light on how simple `Henna`'s processing pipeline is, compared to more complex programs like `switch.p4`. Estimated power consumption is also relatively low. The estimate is based on the *weight* values returned by P4 Insight. In the ingress, `Henna` consumes less than the benchmark, and just 15.56% of the power required by `switch.p4`.



(a) The least consumed resources.

(b) The most consumed resources.

Figure 5.4: Breakdown of the resource usage as a % of the total available in the switch. HBit = Hash Bits, EM Xbar = Exact Match Input Xbar, GW = Gateway, EM SBus = Exact Match Search Bus, and VLIW = Very Long Instruction Words, TR-Bus = Ternary Result Bus, Action Bus = Action Data Bus Bytes, LT-ID = Logical Table ID, TM Xbar = Ternary Match Input Xbar, PHV = Packet Header Vector.

At the egress, power needs are less than half of those of the legacy forwarding program.

Figures 5.4 (a) and (b) provide details on the use of specific switch resources. The figures detail the percent consumption of the total switch capacity by the one-stage model and `Henna`, for each type of resource. The results are separated into two plots based on their magnitude to favour readability across consumption at different scales for two groups of resource types. Even though the usage of most resources is fairly limited, in-switch inference tends to absorb specific types of memory: (*i*) the TCAM and the Ternary Match Input Xbar (TM Xbar) that are used for matching concatenated feature-level codes that include wildcards; and, (*ii*) the PHV used to store the stateless features used for classification through the PISA pipeline. In these cases, `Henna` can consume 30% to 45% of the switch resources. We remark however that these resources are not fully used in legacy forwarding operations, hence the utilization for inference is still compatible with normal functions. Moreover, the high consumption is an inherent limitation of the model mapping, rather than of the hierarchical approach: indeed, the problem affects the single-stage benchmark as well, which only spares 5%–10% of them compared to `Henna`.

5.4. Summary

The main contribution of this chapter is the proposal of `Henna`, a novel per-packet in-switch hierarchical inference solution. The solution is designed as a P4 program that runs on an off-the-shelf Intel Tofino programmable switch. An experimental evaluation based on a device identification use case was performed on the proposed solution and a one-stage benchmark which is representative of previous works. Results shed light on how the hierarchical solution improves classification performance while keeping resource usage at considerable levels.

It is noteworthy that `Henna` is a proof of concept, which only implements a two-stage hierarchy and operates on simple packet-level features. These are limitations that have

to be overcome. To this end, multiple directions are possible. A straightforward one is exploring use cases with multi-stage hierarchies that go beyond the two currently considered; this can be achieved by compressing multiple stages in both the ingress and egress pipelines, as well as by exploiting the multiple pipelines present in high-end programmable switches. An orthogonal approach is to change the classification target of the models from packets to flows, and employ flow-level features for improved model performance; yet, this requires maintaining stateful registers to store flows and stateful features, which is a challenge in constrained switch environments.

Another interesting perspective is extending `Henna` beyond a single switch and distributing different stages or parts of a complex model or ensemble of models over different switches: this would enable distributed inference in the network where different nodes can contribute to a final and possibly more accurate classification result. Also, given that the resource usage footprint of `Henna` in terms of critical resources like PHV and TCAM is fairly high, mainly owing to the RF mapping scheme employed, the exploration of more efficient model mapping techniques is another research direction that will contribute to making in-switch RF deployment more memory-efficient and hence reduce the resource footprint of `Henna`.

Part II

Flow-level inference

Although packet-level inference models are substantially easier to embed in programmable user planes, inference at the flow level is much more relevant in the majority of networking tasks. Generally, flow-level classification provides a more contextual understanding of network traffic [151, 152] by leveraging interesting insights about the relationships between the packets of the same flow. Also, network-wide policies for tasks like traffic engineering [153], or QoS and QoE management [154,155] are generally applied on a flow or service basis, and not at the packet level.

However, deploying models in switches for flow-level inference constitutes a significant challenge as it requires maintaining state and computing flow-level features in resource-constrained devices. This challenge is tackled in this part of the thesis with the proposal and thorough evaluation of a practical solution for flow-level user-plane inference.

6 Flowrest: Practical and general-purpose flow-level inference

In the previous part of this thesis, the promises of in-switch inference based on Decision Tree (DT) and Random Forest (RF) models have been demonstrated at the Packet Level (PL). Inference at PL is intuitive and straightforward given that the switches process packets one at a time, and no complex operations are required to extract and use PL features for inference. This, combined with the strict constraints of programmable switch environments discussed in Chapter 1 has motivated most prior works to focus on PL inference as recently surveyed [30, 31].

Yet, inference at Flow Level (FL) is required for most networking tasks where it is more relevant to consider entire traffic flows[1] than individual packets, *e.g.*, in QoS and QoE management [154, 155]. However, running inference at FL in the user plane is not trivial since it involves storing and computing stateful FL features within the switch. Prior works made attempts to embed FL models in programmable switches for line rate inference but fell short by; only evaluating their proposals in emulated environments which are oblivious of some real-world switch constraints [41, 42]; focusing only on a specific inference task [44, 97, 98]; or not providing implementation details on how stateful information is stored and managed in the switch [40].

Thus, state-of-the-art approaches for in-switch inference at PL or FL have shortcomings in terms of; (*i*) design, *e.g.*, by relying only on basic features extracted independently from the headers of each packet, and lacking flow statistics that are instrumental for effective inference such as inter-arrival times or FL counts; (*ii*) scalability, *e.g.*, of the machine learning model size, and of the complexity of tasks it can address effectively; (*iii*) generalizability *i.e.*, the ability to support multiple use scenarios and applications; and (*iv*) practical viability, *e.g.*, due to incompatibilities with real hardware. As a result, there is a large room for improvement in the design of practical, scalable, and general-purpose solutions for running tree-based Machine Learning (ML) models at FL in real-world programmable switches.

[1] In this thesis, a flow is defined as a sequence of packets with the same 5-tuple of source and destination IP addresses and ports, as well as the same protocol.

This chapter outlines a pioneer and complete design of a practical and general-purpose framework for flow-level inference in programmable switches with random forests, *i.e.*, `Flowrest` [2], which closes the gap above. The proposed solution enables the integration of RF models in production-grade programmable switch hardware to perform challenging inference tasks on individual traffic flows at line rate. Through the development of `Flowrest`, this chapter sets forth the following main contributions.

- A pragmatic framework is proposed to embed generic FL ML models into commercial programmable switch ASICs based on the Protocol Independent Switch Architecture (PISA) architecture. To this end, solutions are proposed to (*i*) run the legacy forwarding logic of the switch and the line-rate inference process simultaneously and in a synergic manner, and (*ii*) effectively compute, store, and exploit stateful per-flow information that is beneficial to complex inference tasks.

- Original guidelines are laid out for the generation of RF models that are hardware-friendly, *i.e.*, natively tailored to the requirements of commercial programmable switches. The strict constraints of network hardware, *e.g.*, of low available memory and limited support for mathematical operations are accounted for by design with workarounds designed to accommodate them during the phases of (*i*) feature engineering and (*ii*) model hyper-parameter tuning.

- The proposed FL framework can accommodate any feasible mapping of DT or RF models onto PISA-based pipelines. The complete solution for in-switch flow-level RF-based inference, dubbed `Flowrest` is first implemented by reproducing and integrating the state-of-the-art RF mapping scheme described in Chapter 3. Then, the ability to accommodate any mapping is demonstrated in Chapter 7 by integrating three other RF model mapping schemes into `Flowrest` and testing them.

- `Flowrest` is implemented into a real-world Intel Tofino switch using the P4 language, alongside five existing approaches in the literature to enable a thorough comparison against the state-of-the-art. The implementation is open source[2], which reduces the significant barriers that exist in the access to hardware-level code of solutions for in-switch inference since most previous works do not provide full implementation details or public code.

Overall, these contributions advance the state of the art in in-switch inference, and make steps towards a deeper integration of ML into network equipment.

[2]The `Flowrest` source code is available on GitHub at https://github.com/nds-group/Flowrest

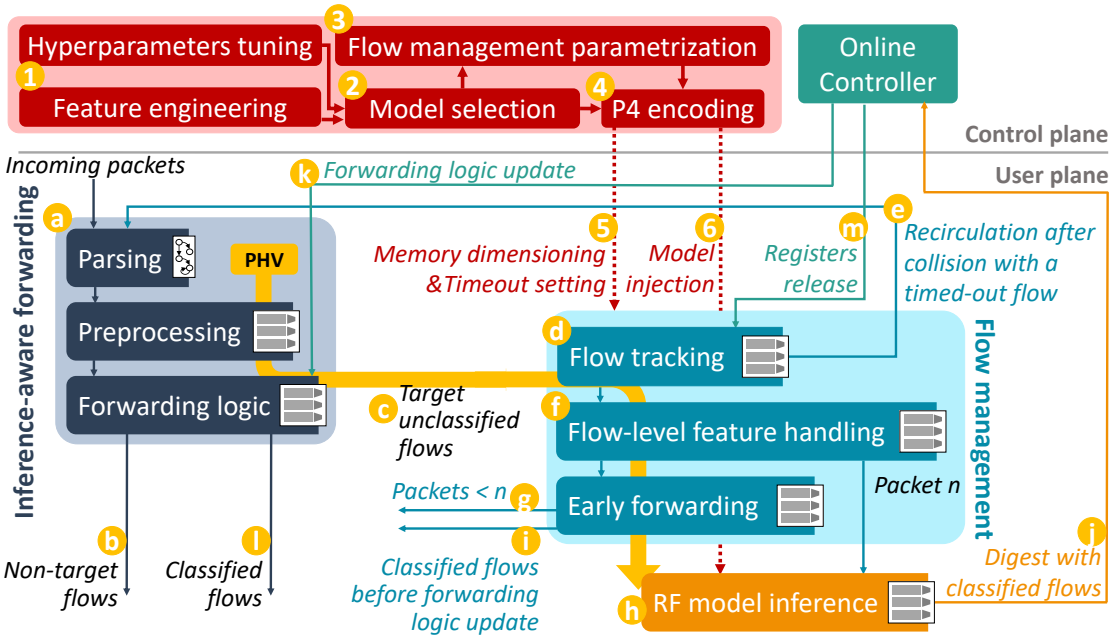


Figure 6.1: Overview of the system proposed for `Flowrest`.

6.1. Flowrest in a nutshell

The system proposed for `Flowrest` is illustrated in Figure 6.1 as a complete framework that enables line-rate classification of traffic flows in the user plane. The components of the framework can be broadly divided into two groups based on whether they are lodged in the control plane or in the user plane. All the compute-intensive operations required for preparing the ML models, *i.e.*, ① - ⑥, are done in an offline server or in the control plane. Then for line-rate inference, all the operations required to parse packets, compute stateful FL features and use them for inference, *i.e.*, ⓐ - ⓙ, are run in the user plane.

In step ①, a grid search for hyperparameters is performed to select hyperparameters and features that will generate models which are tailored to the inherent constraints of programmable switch hardware. The feature and model selection follows the algorithm described in Chapter 3 but with some enhancements including an exhaustive search of the model space, which is not presented in this thesis. Once the best model is identified from the grid search in Step ②, a flow management simulator which mimics the behaviour of the switch pipeline is used to dimension switch memory in Step ③. The design of the simulator is not part of this thesis but is presented in detail in the paper describing the complete `Flowrest` framework [6]. The selected model is then encoded into Match & Action Table (MAT) entries in Step ④ before being injected into the switch in Steps ⑤ and ⑥ alongside the switch configuration settings from the simulator. The control plane also lodges a Python controller which interacts with the user plane via the Barefoot Runtime Interface (BRI).

In the user plane, incoming packets traverse the inference-aware forwarding block in Step ⓐ, which is detailed in Section 6.2.2. Here, packets that are not a target of the inference model or which belong to already classified flows are forwarded in Step ⓑ with no further action. Packets that are part of the target traffic but belong to unclassified flows instead into the flow management block in Step ⓒ.

To classify flows, `Flowrest` exploits FL features calculated over the first n packets of each flow. Each new flow traversing the switch is first identified and tracked in ⓓ with a mechanism based on Cyclic Redundancy Check (CRC) hashes of its flow 5-tuple that assigns a specific memory slot (flow index), and an identifier (flow identifier), as detailed in 6.2.3. In case the newly identified flow finds its assigned slot occupied by another flow, a hash collision event occurs and the new flow cannot be stored and tracked. To minimize the chances of missing the new flow after a collision, a timeout mechanism is introduced to clear stale flow information from the stateful registers after the packet Inter-Arrival Time (IAT) of the flow exceeds a given threshold. In such a case, as depicted by ⓔ, the first packet of the new flow is recirculated to the beginning of the pipeline, since the switch does not allow access to the same register more than once per packet. When it passes through the pipeline after the former colliding flow has been removed from the memory, it can then be stored, tracked, and classified.

In Step ⓕ, the FL features are updated after each new packet of the same flow arrives. The packets arriving before the n-th packet are forwarded with standard rules in Step ⓖ, as they do not have an assigned class yet. When the n-th packet arrives, it goes through the inference module to be classified in Step ⓗ. Upon classification, two actions are taken. First, an *early-forwarding* register is updated with the class of the flow, so that all its subsequent packets can be directly forwarded in ⓘ, avoiding reclassification. Second, a packet digest is sent to the controller in Step ⓙ with the class information, as detailed in 6.2.3. Upon receiving the digest, the controller takes note of the class assigned to the flow and updates the forwarding logic of block ⓐ by sending table updates as depicted by ⓚ. This ensures that newly arriving packets of the flow can be forwarded in Step ⓛ according to a predefined rule. Finally, in Step ⓜ, the controller releases the registers allocated for the classified flow to create room for incoming flows.

6.2. Practical flow-level in-switch inference

`Flowrest` is a complete system for user-plane FL inference, that builds on RF models and user-plane programmability. The main components of the system are lodged in the programmable switch, but `Flowrest` also leverages some control plane functions via, *e.g.*, an ONOS controller to allow for updates to the switch MATs after inference. This section provides details of the `Flowrest` system, highlighting its original aspects.

6.2.1. Parsing

Packets arriving at the switch are processed by its pipeline comprising a parser followed by a sequence of Match & Action Unit (MAU) elements. The parser extracts header fields and metadata from each packet. While the metadata is only functional to forwarding decisions in standard switch operations, in `Flowrest`, the parser acts as a feature extraction module which collects PL information that is relevant to the target inference task. This metadata is stored in the Packet Header Vector (PHV), *i.e.*, a set of containers that carry raw header fields of interest (*e.g.*, the packet size) and gathered metadata (*e.g.*, the packet arrival timestamp) along the whole MAU pipeline.

6.2.2. Inference-aware forwarding

After parsing, packets then enter the Match & Action (M/A) pipeline and first go through the MAUs implementing the legacy behavior of the switch, *i.e.*, the dedicated pre-processing of the metadata required by advanced forwarding rules, and the actual forwarding logic. In `Flowrest`, *the inference-related configurations are integrated into the forwarding logic itself* in the inference-aware forwarding block labelled ⓐ in Figure 6.1, and the controller specifies the following elements.

- The target for inference, *i.e.*, the part of traffic concerned with the inference task. For example, the network manager may target a specific range of source IP addresses suspected to host malicious activities, or a set of protocols known to support the traffic to be classified. The traffic filtering is implemented as a MAT that is part of the forwarding logic, matching packet 5-tuples and tagging packets that should go through in-switch inference logic.

- The status of packets that belong to the inference target group, *i.e.*, whether the flow they are part of has already been classified or not. If yes, the controller configures the switch to deactivate the flag above, as inference is no longer needed.

- The forwarding decision for already classified flows. The controller configures the forwarding logic to handle classified flows based on the outcome of the RF model.

The forwarding logic directly acts upon flows that are not targeted for inference, or for which RF inference results are available. As exemplified by Step ⓑ in Figure 6.1, regular traffic is directed to the egress ports based on legacy rules, *e.g.*, using the destination IP address, while, *e.g.*, flows already identified as, *e.g.*, adversarial by the inference process are dropped right away as illustrated by Step ⓘ, without repeating the inference process.

Clearly, this requires that the controller is aware of the inference results which it uses to configure the forwarding pipeline. To this end, once a flow has been classified by the RF model, the switch is programmed to inform the controller via a *packet digest*, *i.e.*, a small

and flexible message used exclusively for communications between the user plane and the control plane. Sending a digest is more efficient as it enables sending only the required information to the controller as opposed to copying the entire packet which increases overhead. The digest contains the unique flow identifier introduced in Section 6.2.3 and the associated inference result, *i.e.*, its class. The controller can then leverage the received information to update the forwarding logic, as per Step (k) in Figure 6.1.

While the closed loop with the control plane seems to violate the concept of a pure user-plane inference, it is noteworthy that the communication with the controller is totally off the critical path of the inference process, and does not introduce any additional inference latency. The fact that the forwarding logic is reconfigured by the controller with a significant delay of up to a few milliseconds or higher is not a problem in `Flowrest`. Indeed, until the instant when the forwarding logic is updated with the RF model result for a given flow, all decisions about associated packets are taken fully in the switch, via the flow management routine that will be detailed in Section 6.2.3. Ultimately, the system abides by the specifications of pure in-network computation: it ensures that inference happens at all times at line rate and with very low latency, be it in the flow management MAUs at first or in the forwarding MAUs once they have been updated by the controller for subsequent packets.

As a result, the architectural solution adopted by `Flowrest` in Figure 6.1 lets the forwarding and inference stages operate in synergy. The advantages are significant. Processing is faster since packets that have already been (or do not need to be) classified only go through the regular forwarding logic. Also, the model scales better, as it leverages the forwarding tables to offload already classified flows from the flow management registers that, as seen in Section 6.2.3, are limited in size.

Finally, it is important to note that the system above is not feasible with PL inference where flows are not differentiated and packets are processed independently. Also, it is novel with respect to all previous solutions for FL inference, which are bounded to in-switch operation, and do not set forth a complete system where the RF model inference framework benefits form a deep integration with regular forwarding pipelines.

6.2.3. Flow management

Packets that the forwarding logic tags as inference targets, and for which a class is not yet available in the forwarding pipeline are moved to flow management block. This block deals with identifying and tracking flows and the computation, storage and usage of flow-level information. It comprises the phases of *flow tracking*, *flow-level feature handling* and *early forwarding*, which are executed in sequential MAUs, and which all interact with the *flow management table* illustrated in Figure 6.2. The table stores all data relevant to each flow currently under inference, and is implemented as a set of stateful registers in the Static Random-Access Memory (SRAM), and contains (*i*) the flow identifier, (*ii*) its

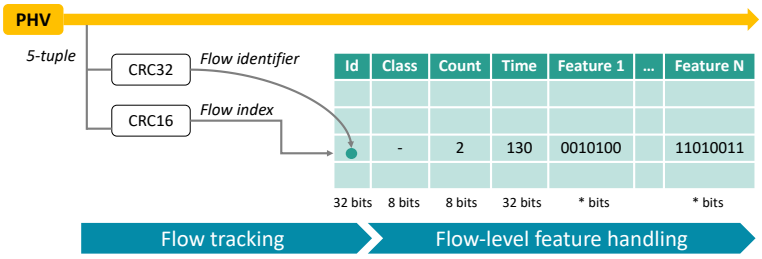


Figure 6.2: Flow management table and its associated stages.

class if already determined, (*iii*) its packet count, (*iv*) the timestamp of the last received packet, and (*v*) all of its stateful FL features.

Flow tracking. This phase is in charge of identifying incoming flows and monitoring their status in the flow management table. First, a *flow identifier* is generated for the packet, via a CRC-32 hash of the 5-tuple composed of source and destination IP addresses, source and destination ports, and the transport protocol identifier. A *flow index* is also computed as a CRC-16 hash of the same 5-tuple and is used to allocate a register row for the flow in the flow management table.

As shown in Figure 6.2, when accessing the flow management table with the flow index, the flow identifier is first compared with what is stored in the table. If the two match or the entry is empty, then the register row is usable and the packet progresses to the flow-level feature handling phase. A mismatch, denoting a hash collision, prevents storing stateful information and thus no inference is performed on the flow at that time, and its packets are recirculated as described in Section 6.1.

Flow index collisions are highly undesirable events that impact the overall performance of the in-switch inference process. Unfortunately, the limited size of registers and the large number of flows traversing a switch may make such events frequent[3]. Previous proposals to cascade hashes in emulated environments [41] are very expensive to implement in hardware as they would require multiple dedicated M/A stages, and so `Flowrest` does not employ them. Instead, the seamless integration of inference with the forwarding logic outlined in Section 6.2.2 naturally mitigates collisions: by offloading the inference-based decision to the regular forwarding pipeline for a given classified flow, the corresponding flow entry can be removed from the flow management table. This limits the number of flows concurrently present in the table and dramatically reduces collisions[4]. A timeout is also employed to remove flows from the flow management table. Every time a collision occurs, the time elapsed since the arrival of the last packet of the flow already in the table is checked: if it is above a threshold[5], the flow data is purged from the registers and the register row its index points to is then able to accommodate the new flow.

[3]In the targeted use cases, more than 10% of flows collide without mitigation.
[4]In the targeted use cases, the collision probability is nearly at 0% with `Flowrest`.
[5]Timeout thresholds of 256 milliseconds to 15 seconds worked well in the different use cases analyzed.

Flow-level feature handling. Once packets pass the flow tracking phase, the FL features are computed, updated and stored in the flow management table. As shown in Figure 6.2, each entry contains the class of the flow if available, the count of recorded packets in the flow, and a timestamp of the arrival of the last packet used for the computing time-related features and checking for timeout as described above. In addition, it stores all the stateful features needed by the FL RF model. The exact list of features depends on the specific RF model, and, in the experiments conducted, it is always a subset of the complete flow-level features listed in Chapter 3. Several aspects of this phase are noteworthy and detailed next.

- From a technical standpoint, all the stateful variables in the flow management table must be read and updated simultaneously, which is a non-trivial operation in hardware. This is achieved in `Flowrest` via the Tofino Native Architecture (TNA) RegisterAction extern that exploits the Stateful Arithmetic Logic Unit (S-ALU) associated with the registers for mathematical and logical operations. For each feature, the data is fetched from the respective register and computed by executing an action associated with the register, which uses the data retrieved from the packet it is processing.

- The packet count is used to implement an *early-flow detection* approach [86], where `Flowrest` performs inference upon reception of the first few packets of each flow as also done by pForest [41]. Only a single n-th packet in the sequence of flow packets[6] is processed by the RF model; this is a further advantage over PL inference where the RF model must process all packets in each flow. This is also much more responsive than a full-flow detection where statistical features associated with a traffic flow are computed over a larger number of packets, *e.g.*, in SwitchTree [42].

- The class indicates the classification outcome of the RF model for the flow. To store this information, two RegisterActions; *read* and *update* are used. After the n-th packet has completed the inference pipeline, the *update* action is triggered and this stores the class of the packet in the dedicated register. If the packet is not the n-th one, the *read* action is instead executed to check if the flow already has an assigned class based on which inference can be skipped. Using these two mutually exclusive actions avoids performing recirculation as previously done in an earlier version of `Flowrest` [2] and reduces latency.

Early forwarding. By coupling an early-flow detection and the preliminary storing of the classification outcome, `Flowrest` can immediately react to the outcome of the inference process, which improves the speed of response without sacrificing quality. Once

[6] In the experiments conducted, early-flow classification is conducted at the second, third or fourth packet, *i.e.*, $n \in \{2, 3, 4\}$.

a packet reaches the flow management stage and is found to belong to an already classified flow, it is directed to the early forwarding stage. Here, a dedicated logic is implemented for already classified flows, which takes forwarding decisions based on the class reported in the flow management table. Figure 6.1 shows how, *e.g.*, all packets beyond the n-th of a flow classified are directly forwarded at this stage as portrayed by Step ①.

As mentioned in Section 6.2.2, the early forwarding is controlled by the flow management table for each recently classified flow, until the controller is informed of the flow class and the main forwarding logic is updated to reflect that. The early forwarding phase is also configured to handle packets *before* the n-th in each flow. While they contribute to flow-level statistics in the feature handling phase, these packets come too early in the flow to be classified; passing them on to the early forwarding ensures that they are processed, *e.g.*, based on standard rules.

6.2.4. RF model mapping

When the flow management processes the n-th packet of a target unclassified flow, the up-to-date FL features stored in the PHV alongside the PL ones are employed to execute an RF model encoded in the subsequent stages of the M/A pipeline, as shown in Figure 6.1. The framework developed for `Flowrest` can accommodate any hardware-feasible model or mapping strategy without changes to the workflow of the stages presented before. To the best available knowledge, `Flowrest` is the first framework for FL in-switch inference that allows for easy integration of different models and mapping strategies. To build a complete solution for in-switch FL RF-based inference, the custom version of the original Planter RF model mapping detailed in Chapter 3 is used. Then subsequently, three other mapping strategies are integrated to demonstrate the flexibility of the framework.

6.2.5. Switch-tailored model design

A distinguishing aspect in the operation of `Flowrest` is the fact that *hardware constraints are accounted for right from the ML model preparation phase.* This facet of `Flowrest` is illustrated by the top portion of Figure 6.1 and comprises all the steps involved with model preparation, numbered ① – ④, and which are implemented in the control plane and run offline. The hardware-tailored modelling process stems from the way flow management and RF inference mapping are implemented, and ensures that the trained RFs are compatible with the programmable switch architecture *by design.*

As described in Chapter 3, the popular Scikit-Learn libraries [73] are used for the model preparation stage, while tailoring the feature engineering and selection of hyperparameters to accommodate implementation into real-world equipment. The complete ML modelling pipeline is organized into two phases; feature extraction and engineering and model design. These phases are described next.

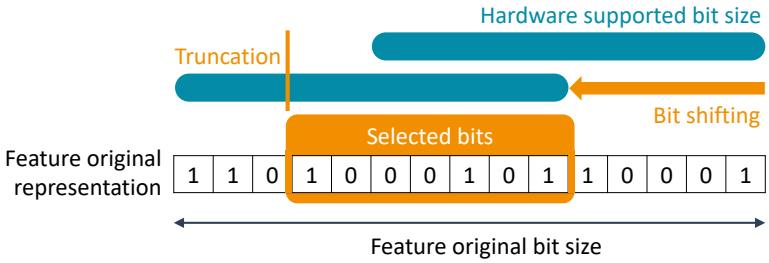


Figure 6.3: Hardware-aware tuning of feature representation.

Feature extraction and engineering. Both the PL and FL features presented in Table 3.1 are computed from the training data using Tshark and Python as described in Chapter 3. The feature and model selection process follows Algorithm 1. What makes the feature engineering unique in `Flowrest` is that *the representation of the features are adapted to the specifications of the hardware*. More precisely, state-of-the-art RF mapping strategies use range matches to compare feature values against thresholds from the model, as explained in Chapter 3. The size of range match keys is bounded to a maximum number of bits in real-world equipment[7]. While this limit hardly affects PL features rendered over a few bits, it affects the FL ones, particularly time-related features that are represented by default with floating point precision.

To cope with this issue, `Flowrest` fine-tunes the description of features exceeding the size limit, by (*i*) shifting their binary representation, and then (*ii*) truncating it to only consider an ideal bit length smaller than the limit. This procedure, illustrated in Figure 6.3, happens at training time via an exhaustive search of the best combination of shift and truncation for each feature to be compressed. For example, if a feature is 64 bits wide, it cannot fit into the ternary match crossbars used to store the match keys. `Flowrest` loops over the feature values, beginning with only one bit and adding bits progressively from the most significant to the least significant, and noting the resulting model performance. The shift and truncation that yield the best result are adopted and used when training the final model. The result can be replicated in hardware during the flow-level feature handling phase, as it only uses basic binary operations. The fine-tuning ensures that all comparisons used by the RF model are compatible by design with the capabilities of the target programmable switch and that training is optimized for performance in real-world equipment.

Model design. As in Algorithm 1, a grid search is performed on the maximum number of trees t, and the maximum depth of each tree d. Therefore, the feature ranking and processing described in Chapter 3 is derived for each RF with hyperparameters t and d. The models with all (t, d) combinations are compared based on the F1-score defined in Chapter 3, and the best performing one is picked. The model size ultimately depends on

[7]The exact maximum size depends on the equipment and is confidential under Intel NDA.

the complexity of the use case: for instance, in the use cases considered for the evaluation in Chapter 7, d is in the range $[3, 20]$ and t is in the range $[1, 5]$.

Importantly, *the choice of RF model hyperparameters is also tailored to the underlying hardware target.* Specifically, state-of-the-art RF mapping strategies such as that presented in Chapter 3 use ternary matches on codewords and associated masks to perform the final classification. This implies that the codeword length cannot exceed the maximum ternary match key size allowed by the actual Ternary Content-Addressable Memory (TCAM) blocks present in the switch. `Flowrest` accounts for the constraint above during the design phase. To this end, the fact that each tree in the RF generates one codeword, where every bit encodes one node of the RF, leaf nodes excluded is leveraged.

A full binary tree with $n - 1$ total nodes will always have n leaves. Knowing this, the codeword length can be controlled by limiting the number of allowed leaves to the maximum ternary match key size in the programmable switch. This design control is implemented by selecting the best nodes in each tree based on the relative reduction in impurity [156], until the target number of leaves is attained. This is achieved in Scikit-Learn using the *max_leaf_nodes* parameter of the RF classifier object. It is noteworthy that limiting the number of leaves does not place any limits on the tree depth, which can span to $d = n$, so that the model retains its flexibility.

6.3. Summary

This chapter presented a practical and general-purpose framework for FL inference in programmable switches named `Flowrest`. Original guidelines were presented for preparing RF models that are hardware-friendly by taking switch constraints into account right from the design phase. `Flowrest` sets a new standard for FL in-switch inference, yielding a number of technical contributions and novel concepts that have general application to user-plane ML, like the synergy of user and control planes for effective line-rate inference, or the design of hardware-aware ML models. These concepts go beyond the specific `Flowrest` solution and may open new research pathways in the domain.

7 Flow-level inference application case studies

As introduced in Chapter 6, Flow Level (FL) inference is ideal for a variety of in-switch inference use cases. To demonstrate the capabilities of the proposed `Flowrest` framework, a thorough experimental evaluation is conducted in this chapter. The evaluation begins with five unencrypted traffic classification case studies, which reveal how `Flowrest` supports various use cases, outperforms existing work, and is adaptable to other model mapping schemes. Then its ability to classify encrypted traffic is showcased in the second part of the evaluation with three case studies.

7.1. Unencrypted traffic classification

Majority of Internet-of-Things (IoT) traffic is unencrypted [157]. This exposes IoT devices and networks to many security vulnerabilities since unencrypted traffic constitutes a juicy target for malicious activity. As such, most use cases explored in existing work on user-plane inference are based on IoT-related datasets [78, 119, 120] which generally contain unencrypted traffic for use cases like intrusion detection and device fingerprinting.

`Flowrest` and five benchmark solutions are implemented in the switches of the hardware testbed described in Chapter 3, and experiments are conducted with multiple user-plane inference use cases. Tasks of unprecedentedly high dimensionality are included to assess how the performance of a practical flow-level approach scales with the complexity of the inference problem.

7.1.1. Use cases

The inference use cases targetted include: (*i*) device identification based on the UNSW-IoT traces [78]; (*ii*) IoT bot classification inspired by the IoT-23 dataset [119]; (*iii*) IoT cyberattack classification based on the ToN-IoT telemetry data [120]; (*iv*) service classification with the UNIBS-2009 Internet traces [79, 80]; and (*v*) intrusion detection which builds on the CICIDS-2017 dataset [81]. They are detailed in Chapter 3.

7.1.2. Benchmarks and levels of comparison

To establish `Flowrest` as the new standard for in-switch FL inference, it is compared to five benchmarks. These benchmarks are discussed alongside other related work in Chapter 2. Their implementation details are presented next.

Planter [43]. It is representative of recent proposals like Planter [40], IIsy [38], Henna [4], and Bütün et al. [10] which re-use its mapping scheme. The source code provided in [158] is employed to reproduce this benchmark.

Mousika [93]. The models are implemented with the Mousika mapping using the publicly available code [159]. Following the workflow of the original model, each of the sets of selected features for both the Packet Level (PL) and FL models is binarized and used to train a teacher Random Forest (RF) whose prediction probabilities are then used to train a student BDT. The resulting BDT is then mapped onto the switch pipeline.

pForest [41]. The source code is not yet publicly available so a custom version of the mapping scheme is implemented. The comparison logic at tree nodes is not trivial to implement and requires an additional stage. This means for trees of maximum depth N, the pForest implementation requires $N \times 2$ stages to implement trees in parallel, and an additional stage for deciding the final result of the RF from the individual tree results. With only 12 Match & Action (M/A) stages available on the switch, for PL models with simpler logic and no additional computations requiring more stages, RFs of maximum depth 5 can be implemented. In FL models, $3-5$ M/A stages are required for the flow management and RF voting steps. Thus, only 7 stages are available for tree levels, only allowing trees of maximum depth 3 for pForest, which represents other works that employ the same tree mapping technique like BACKORDERS [94] and SwitchTree [42], both of which are not tested in hardware.

Soter [96]. It uses a similar approach as pForest [41], mapping each level of a tree to one M/A stage but overcomes the need for additional stages for logical operations by using Ternary Content-Addressable Memory (TCAM)-based range matches to compare the feature thresholds and feature values at tree nodes. The publicly available code [160] is used as a starting point, and then it is extended to enable (*i*) support for RFs, (*ii*) other use cases, and (*iii*) FL inference with `Flowrest`. In the PL models, RF models with maximum depths of up to 10 can be supported, with the extra stage reserved for voting. In the FL models, subtracting 4 stages for flow-related computations, only trees of maximum depth up to 6 can be supported. The models in this chapter are tailored to these constraints, often resulting in the choice of a smaller model for this benchmark and hence, a drop in accuracy compared to other mappings.

NetBeacon [101]. The publicly available source code [161] is employed for the implementation of this benchmark. It should be noted that although the paper describes how the proposed model mapping scheme can map RF models with multiple trees, this

capability is neither clearly shown in the model design, nor is it illustrated in the released artifacts [161] where only single-tree RFs are used, even for a use case where the RF was indicated to have more trees. As it is not obvious how the model can be extended to map multiple trees, the models evaluated in this chapter are restrained to RFs with only 1 tree for this benchmark.

The evaluation is conducted at three levels by comparing `Flowrest` to existing (*i*) PL solutions, (*ii*) FL and hybrid solutions, and (*iii*) demonstrating that `Flowrest` can accommodate different RF mapping schemes.

(*i*) **`Flowrest` vs packet-level solutions**. `Flowrest` is compared against three PL solutions namely Planter [43], Mousika [93] and Soter [96]. For these PL RF models, dedicated RFs are trained using only PL features with Scikit-Learn libraries, and they are implemented in hardware by stripping off all the FL operations from `Flowrest`. This evaluation sheds light on how `Flowrest` outperforms stateless PL solutions.

(*ii*) **`Flowrest` vs flow-level solutions**. `Flowrest` is compared to a reproduced version of pForest [41], and the recent hybrid packet-flow classifier, NetBeacon [101], to show how `Flowrest` either outperforms or is on par with other stateful classifiers, consuming fewer resources even in the case where classification performances are similar. FL models are trained starting with the same feature set and then implemented in the different approaches.

(*iii*) **`Flowrest` with different RF mappings**. The ability of the `Flowrest` framework to accommodate any feasible RF mapping as detailed in Chapter 6 is demonstrated by implementing `Flowrest` in 3 additional mappings proposed by Mousika [93], Soter [96], and pForest [41], in addition to the Planter [43] mapping already employed in `Flowrest`. An RF is trained with the same data and features and then mapped into the switch using all 4 mappings embedded into the `Flowrest` framework. The NetBeacon [101] mapping is left out at this level of evaluation since its ability to map RFs is still unclear.

7.1.3. Results and discussion

The performance of `Flowrest` and the benchmark solutions are assessed in terms of classification accuracy, switch resource usage, and packet-processing latency.

7.1.3.1. Classification accuracy

The classification performance of `Flowrest` and the benchmarks are evaluated using three metrics; precision, recall, and F1-Score. For the PL models, their performances at FL are obtained using the normalization by flow length introduced in Chapter 3.

The accuracy of `Flowrest` is compared to those of the PL solutions in Table 7.1. Overall, `Flowrest` models outperform their per-packet counterparts in flow classification.

Dataset	Average	Metric	Planter	Mousika	Soter	Flowrest
CICIDS	Macro	Precision	94.448%	87.920%	94.446%	**99.785%**
		Recall	92.900%	78.359%	92.906%	**98.682%**
		F1-Score	93.625%	81.231%	93.628%	**99.231%**
	Weighted	Precision	94.712%	86.668%	94.713%	**99.700%**
		Recall	94.734%	86.009%	94.74%	**98.556%**
		F1-Score	94.688%	85.015%	94.690%	**99.124%**
UNIBS	Macro	Precision	82.561%	86.437%	83.332%	**96.494%**
		Recall	86.183%	86.752%	86.765%	**97.473%**
		F1-Score	83.859%	86.281%	84.360%	**96.911%**
	Weighted	Precision	97.465%	98.802%	97.578%	**99.602%**
		Recall	95.930%	98.247%	95.940 %	**99.521%**
		F1-Score	96.489%	98.500%	96.552%	**99.552%**
IoT23	Macro	Precision	**94.513%**	90.873%	73.912%	93.458%
		Recall	79.191%	87.030%	61.557%	**90.331%**
		F1-Score	84.445%	88.542%	61.943%	**91.621%**
	Weighted	Precision	98.791%	**99.389%**	97.997%	99.315%
		Recall	98.788%	**99.394%**	97.817%	98.577%
		F1-Score	98.693%	**99.386%**	97.662%	98.910%
UNSW	Macro	Precision	54.822%	67.882%	53.608%	**72.839%**
		Recall	57.523%	80.543%	55.677%	**81.760%**
		F1-Score	48.502%	69.103%	47.498%	**72.277%**
	Weighted	Precision	78.597%	90.166%	78.329%	**91.538%**
		Recall	73.906%	88.285%	72.208%	**89.165%**
		F1-Score	73.055%	88.572%	73.084%	**89.733%**
ToN-IoT	Macro	Precision	54.208%	28.728%	51.631%	**68.425%**
		Recall	43.866%	44.916%	41.590%	**59.332%**
		F1-Score	44.711%	30.882%	41.500%	**60.494%**
	Weighted	Precision	44.111%	40.376%	43.671%	**86.067%**
		Recall	44.258%	38.330%	40.947%	**84.928%**
		F1-Score	41.080%	37.463%	39.887%	**85.303%**

Table 7.1: Performance of `Flowrest` compared to the P-L benchmarks. The best value on each row is in bold.

Considering the macro precision, `Flowrest` achieves absolute and relative gains over the second-best model for the CICIDS, UNIBS, UNSW and ToN-IoT datasets in the ranges of $1.84\% - 14.22\%$ and $1.94\% - 26.23\%$ respectively, falling short only in the IoT23 dataset where the Planter benchmark slightly outperforms `Flowrest` by 1.06% on absolute terms and 1.12% relatively. The gains of `Flowrest` over the benchmarks are even bigger when considering the weighted precision, going up to 95.11% in the ToN-IoT dataset. This better precision means that `Flowrest` achieves the best quality of positive predictions when compared to the benchmarks.

`Flowrest` also achieves huge gains in terms of recall which implies that `Flowrest` returns fewer false negatives than the per-packet benchmarks, which is important in

Dataset	Average	Metric	pForest	NetBeacon	Flowrest
CICIDS	Macro	Precision	99.778%	98.251%	**99.785%**
		Recall	98.690%	**98.918%**	98.682%
		F1-Score	**99.231%**	98.576%	**99.231%**
	Weighted	Precision	99.697%	98.816%	**99.700%**
		Recall	98.556%	**98.793%**	98.556%
		F1-Score	99.123%	98.798%	**99.124%**
UNIBS	Macro	Precision	48.612%	86.905%	**96.494%**
		Recall	53.667%	90.986%	**97.473%**
		F1-Score	50.894%	88.649%	**96.911%**
	Weighted	Precision	94.058%	97.854%	**99.602%**
		Recall	93.966%	97.190%	**99.521%**
		F1-Score	93.934%	97.404%	**99.552%**
IoT23	Macro	Precision	37.294%	85.272%	**93.458%**
		Recall	29.066%	80.766%	**90.331%**
		F1-Score	30.605%	81.528%	**91.621%**
	Weighted	Precision	87.706%	99.022%	**99.315%**
		Recall	88.406%	**99.018%**	98.577%
		F1-Score	86.439%	**98.941%**	98.910%
UNSW	Macro	Precision	14.183%	56.256%	**72.839%**
		Recall	18.412%	66.089%	**81.760%**
		F1-Score	15.200%	53.284%	**72.277%**
	Weighted	Precision	41.582%	81.261%	**91.538%**
		Recall	48.407%	73.394%	**89.165%**
		F1-Score	43.034%	75.470%	**89.733%**
ToN-IoT	Macro	Precision	35.008%	52.521%	**68.425%**
		Recall	32.601%	43.800%	**59.332%**
		F1-Score	31.022%	43.977%	**60.494%**
	Weighted	Precision	71.682%	47.172%	**86.067%**
		Recall	72.252%	45.116%	**84.928%**
		F1-Score	69.019%	40.921%	**85.303%**

Table 7.2: Performance of `Flowrest` and the flow-level benchmarks. The best value on each row is in bold.

security-related use cases. When recall is combined with precision to calculate the F1-score, `Flowrest` outperforms all the per-packet benchmarks in all use cases, considereing the macro F1-score, with absolute gains of up to 15.78% and relative gains of up to 35.30%. When weighted, the gain in F1-score is as high as 44.223% on absolute terms and 107.65% on relative terms. Ultimately, these results validate the superiority of the FL classification enabled by `Flowrest` over existing PL solutions.

The performance of `Flowrest` is also compared to those of other stateful solutions, *i.e.*, pForest [41] and NetBeacon [101] in Table 7.2. The results show that in relatively easy tasks like CICIDS where simple models are good enough to solve the classification task, all the solutions perform similarly, with `Flowrest` having only a slight gain over the others

Average	Metric	pForest	Soter	Mousika	Planter
Macro	Precision	99.778%	**99.785%**	99.770%	**99.785%**
	Recall	**98.690%**	98.678%	98.652%	98.682%
	F1-Score	**99.231%**	99.228%	99.208%	**99.231%**
Weighted	Precision	99.697%	99.699%	99.683%	**99.700%**
	Recall	**98.556%**	98.550%	98.536%	**98.556%**
	F1-Score	99.123%	99.121%	99.106%	**99.124%**

Table 7.3: Performance of `Flowrest` when RF models for the CICIDS dataset are encoded with different mapping schemes. The best value on each row is in bold.

in terms of F1-score. When the use cases become more complex and require larger RF models, the gains of `Flowrest` over the others become more significant in terms of all metrics in the UNIBS, UNSW and ToN-IoT datasets, with absolute gains of up to 18.99% and relative gains of up to 27.30%.

In the case of pForest, the main reason for the drop in performance with increasing use case complexity is the limitation in the maximum depth of trees in the RFs as explained in Section 7.1.2. As only models with a maximum depth of 3 could be mapped, all the models predicting more than 8 classes have low scores since there are more classes than tree leaf nodes. In contrast, `Flowrest` models have a maximum depth of up to 20 and thus ensure better performance and scalability. In NetBeacon, a PL model is the fall-back model in all cases where a FL classification cannot be made, *e.g.*, in the case of a hash collision, when too few packets have arrived, or when the flow is predicted as short. This means that a large number of packets use this model which is generally less accurate and so pulls down the overall score. Instead, `Flowrest` only performs FL classification with a well-tailored flow management scheme to deal with hash collisions, ensuring high accuracy. Additionally, as the deployed RF models in NetBeacon only have one tree, they are unlikely to be as accurate as the multi-tree RFs of `Flowrest`.

Lastly, using the CICIDS dataset which requires relatively simple models, it is demonstrated in Table 7.3 that any feasible RF mapping can be integrated into the `Flowrest` framework for FL inference. All the models achieve very similar classification results, with slight differences mainly due to the approximations involved with each mapping scheme. This establishes `Flowrest` as a viable tool for converting any PL inference system into a FL system.

7.1.3.2. Resource usage

The resource footprint of the in-switch models are assessed with the help of the P4 Insight tool [116] and the results are presented in Figure 7.1. The resource usage is evaluated in terms of key resources; TCAM and Static Random-Access Memory (SRAM), and the average of all other resources. `Flowrest` consumes less TCAM than all the PL benchmarks in the CICIDS and ToN-IoT datasets and is the second least consumer in the

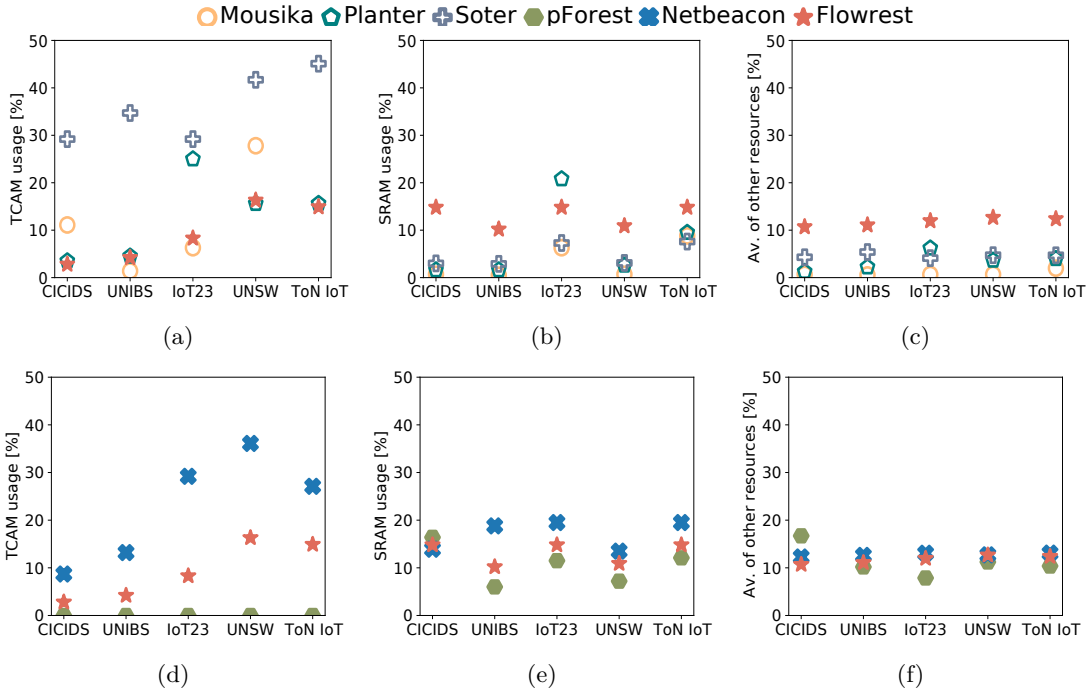
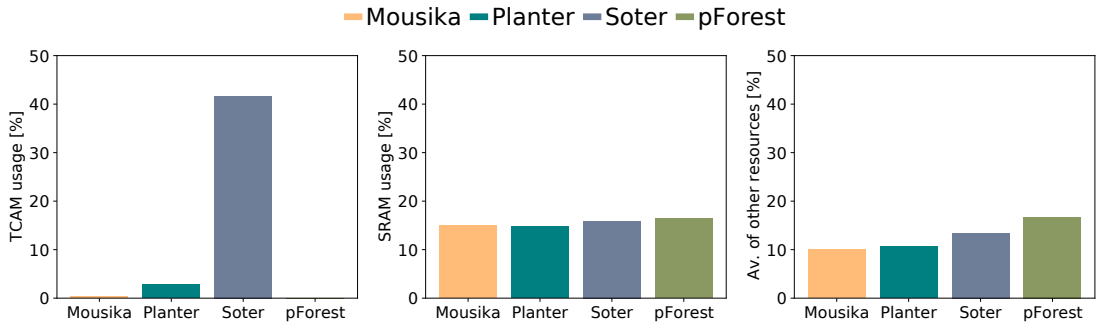


Figure 7.1: Comparison of resource usage of `Flowrest` vs per-packet benchmarks in terms of (*a*) TCAM; (*b*) SRAM; (*c*) average of the remaining resources; `Flowrest` vs per-flow benchmarks in terms of (*d*) TCAM; (*e*) SRAM; (*f*) average of the remaining resources.

UNIBS, IoT23 and UNSW datasets, as shown in Figure 7.1(a). Comparing also against the FL solutions in Figure 7.1(b), `Flowrest` always consumes much less TCAM than NetBeacon, with pForest not consuming any, while suffering from scalability issues. This demonstrates the resource efficiency of `Flowrest` in terms of TCAM which is the most expensive resource.

With regards to SRAM, `Flowrest` generally consumes less than NetBeacon but a bit more than pForest as shown in Figure 7.1(d). In addition, it also consumes more SRAM than the PL benchmarks due to the need to maintain stateful registers for traffic flows. The only exception is the Planter benchmark in the IoT23 dataset where the model has a very large voting table with exact matches which use up SRAM. Considering the average of all the other resources shown in Figures 7.1(e) and 7.1(f), `Flowrest` consumes more resources than the PL benchmarks due to the need to maintain state, but always consumes less resources than its closest stateful competitor, NetBeacon. Juxtaposing these consumptions with their classification performances, `Flowrest` achieves by far a better trade-off between memory consumption and classification accuracy, always consuming on average less than 15% of switch resources, leaving room for the switch to implement other functions like forwarding the background traffic injected at 40 Gbps with the MoonGen [118] traffic generator.

The ability of `Flowrest` to accommodate any mapping enables comparing the resource footprint of the mapping schemes proposed by pForest [41], Soter [96], Mousika [93],



(a) (b) (c)

Figure 7.2: `Flowrest` versions for the CICIDS dataset in 4 RF mappings in terms of (*a*) TCAM; (*b*) SRAM; (*c*) average of the remaining resources.

	Mousika	Planter	Soter	pForest	NetBeacon	Flowrest
CICIDS	301.64	318.03	386.89	460.66	419.67	436.07
UNIBS	300.82	336.07	457.38	459.02	436.07	436.07
IoT23	300.82	338.52	440.98	460.66	419.67	404.10
UNSW	303.28	302.46	455.74	459.02	421.31	418.03
ToN-IoT	309.84	336.89	440.98	459.02	419.67	418.03

Table 7.4: Estimated packet processing latency in *nanoseconds* of the in-switch models.

and Planter [43]. Figures 7.2(a) and 7.2(c) reveal that while Soter consumes the most TCAM, pForest consumes the most of the other resources, even though it uses no TCAM. Figure 7.2(b) shows that irrespective of the mapping scheme used, the SRAM consumption stays relatively the same. This is because SRAM is mainly used to maintain the stateful information for flow classification and since this is dimensioned by the `Flowrest` framework, it is relatively constant. The results also shed light on the fact that the Mousika mapping generally has the least resource footprint, although its TCAM consumption could sometimes be very high like in the UNSW and ToN-IoT use cases where the generated binary decision tree is very large.

7.1.4. Latency

All the in-switch models run at line rate. The delay in nanoseconds experienced by a packet that goes through the switch and undergoes classification is estimated. P4 Insight provides the estimated packet-processing latency of P4 programs in terms of cycles, considering the ingress and egress M/A pipelines. Knowing the clock frequency of the switch, these values are converted into nanoseconds and presented in Table 7.4. All the in-switch models deployed incur sub-microsecond delay. In the case of `Flowrest`, the latency never exceeds 436 nanoseconds, just about 136 nanoseconds more than the simplest per-packet solution. In addition, when compared to the other FL solutions like pForest and NetBeacon, `Flowrest` typically induces less latency. These results confirm that `Flowrest` can perform line rate inference in the switch, with only hundreds of nanoseconds of delay.

7.2. Encrypted traffic classification

Recent works have demonstrated the feasibility of in-switch Machine Learning (ML) inference with Decision Tree (DT) and RF models for classifying traffic at line rate in programmable switches [4, 10, 43, 93]. However, none of these works specifically targets Encrypted Traffic Classification (ETC), neither do they take into account the possible unavailability of some features due to encryption. This second part of the evaluation fills the above gap by building on `Flowrest` to integrate the ETC process fully into programmable switches to perform line rate inference on traffic flows in real-time using RF models trained offline and then embedded into the switch pipeline [5].

To that end, the workflow proposed for `Flowrest` is re-used with the particularity that the RF models employed are made to be ETC-aware by only extracting and using statistical features based on the packet size and packet Inter-Arrival Time (IAT). These two features are generally unaffected by encryption algorithms [124] and so would be available for ETC most of the time. Port numbers and other header fields that might be affected by dynamic port number schemes and masquerading are excluded. Flow statistics are then computed and the following nine features are considered for in-switch inference; *the maximum, minimum, total* and *mean* of the two base features; *packet sizes* and *IAT*, and *the last observed packet size.* The total IAT corresponds to the duration of the flow.

The features above are used to train and evaluate RF models via a 12-fold cross-validation. The models are then implemented as open-source P4 software[1] in a production-grade Intel Tofino switch. The programs have between $663 - 720$ lines of P4 code, with in-switch RF models of maximum depth in the range $16 - 20$, number of trees in the range $1 - 5$, and $4 - 5$ features retained after the feature selection process.

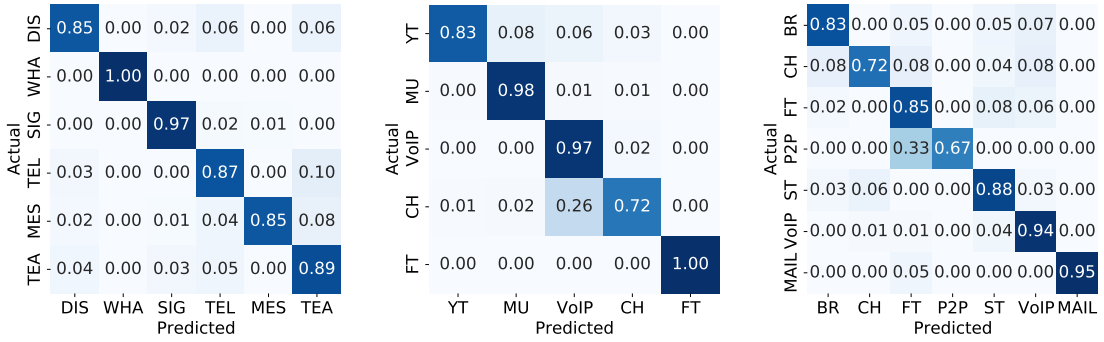
7.2.1. Use cases and datasets

The targeted use cases include; encrypted instant messaging application fingerprinting based on the NIMS IMA dataset [121,122], QUIC traffic classification based on the Netflow QUIC data [123], and Virtual Private Network (VPN) traffic classification based on the ISCX-VPN-NonVPN-2016 dataset [125]. The datasets are described in Chapter 3.

7.2.2. Results and discussion

The performance of `Flowrest` on ETC is assessed via classical metrics notably; (*i*) accuracy, and (*ii*) F1-score, expressed as macro and weighted averages as described Chapter 3. The resource consumption of the in-switch models is also assessed.

[1]The source code is available on GitHub at https://github.com/nds-group/ETC_NOMS_2024.



(a) (b) (c)

Figure 7.3: Classification performance from in-switch ETC experiments showing confusion matrices for (a) NIMS IMA; (b) Netflow QUIC; and (c) ISCX VPN.

7.2.2.1. Classification performance

The classification performance of the in-switch ETC solution over the three use cases is presented in Figure 7.3. Overall the ETC models achieve high classification accuracy in the tasks targeted, starting from 87.16% for the ISCX VPN dataset, which seems to be the most challenging, through 89.74% for the NIMS Instant Messaging Application (IMA) dataset, to 95.28% for the Netflow QUIC dataset, where the highest accuracy is obtained. In terms of the macro F1-score which gives a measure of how good the classifier is irrespective of the number of samples of each class, the scores are also high, ranging from 85.64% in ISCX VPN through 90.29% in NIMS IMA to 90.31% in Netflow QUIC. This shows that although the datasets are generally unbalanced between classes, the ETC models are still able to do well in distinguishing them. Factoring the imbalance into account and considering the weighted F1-scores, the scores are higher, ranging from 87.14% in ISCX VPN through 89.82% in NIMS IMA to 95.29% in Netflow QUIC. The higher weighted F1-scores mean that the models tend to perform very well on the most represented classes and so pull the overall score upwards when the average is weighted by number of samples.

The confusion matrices in Figures 7.3(a), 7.3(b), and 7.3(c) show the classification performance per class and per use case of the scenarios considered. For IMA fingerprinting (NIMS IMA) shown in Figure 7.3(a), the ETC model excels at identifying *WhatsApp*, and *Signal* with 100% and 97% respectively. For all the other IMAs, the ETC model never goes below 85% accuracy. In the case of QUIC traffic classification portrayed in Figure 7.3(b), for the *File Transfer*, *Google Play Music*, and *Google Hangout VoIP* classes, the ETC model achieves over 97% accuracy, coming short only in the *Youtube*, and *Google Hangout Chat* classes where it achieves 83% and 72% respectively. This lower score for the *Google Hangout Chat* class could be attributed to the diverse nature of its traffic, and to the reduced amount of data used for the in-switch tests to allow for experimentation in controlled settings.

Resource	Datasets		
	NIMS [%]	**Netflow QUIC [%]**	**ISCX VPN [%]**
Action Data Bus Bytes	12.4	7	13.9
Logical Table ID	16.1	15.6	15.6
SRAM	4.4	5.4	4.4
TCAM	11.5	7.3	15.3
Ternary Match Input Xbar	18.9	11.1	24.2
VLIW Instruction	6.3	5.5	6.3
Total Avg. Resource Usage	9.09	8.33	9.76
Number of M/A Stages	12	10	12

Table 7.5: Summary of the usage of key switch resources.

The VPN traffic classification (ISCX VPN) case shown in Figure 7.3(c) is the most challenging. Just like in the QUIC classification use case, *Chat* traffic remains one of the hardest to identify with 72% of samples correctly predicted, seconding only *P2P* with just 67% accuracy. In all cases where *P2P* traffic is misclassified, it gets confused for the *FT* class, which is understandable given that there are multiple peer-to-peer file transfer apps. For the rest of the classes, the performance of the ETC solution is quite good, ranging from $83\% - 95\%$, demonstrating that FL ETC is feasible in user-plane switches.

7.2.2.2. Resource usage

The resource consumption of the P4 programs are assessed with P4 Insight and the results are reported in Table 7.5. Overall, the model deployments consume less than 10% on average of the total available resources on the switch, leaving room for cohabitation with other switch functions. Considering individual resources, the models only consume between $4.0\% - 5.5\%$ of SRAM, which is used for the stateful registers that store flow information. This leaves a lot of SRAM for other switch functions and consumption can be further controlled by tuning the number of allocated entries in the registers. The Very Long Instruction Words (VLIW) which are used for mathematical operations, are also only consumed at about $5.5\% - 6.3\%$, leaving room for other applications.

TCAM consumption varies across use cases. In the Netflow QUIC dataset, as the RF has only 1 tree, less TCAM is required and fewer bits of crossbars (Xbar) are needed, explaining the lower values in Table 7.5. The model of the ISCX VPN dataset has trees with more leaves which require more ternary match bits in its tree MATs. That explains the 24.2% of Xbar required for its match keys. Regarding the action data bus bytes that carry action data, consumption is as low as 7% in Netflow QUIC but rises to 12.4% in NIMS and 13.9% in the ISCX VPN case due to trees having more leaves hence, longer code words. The P4 programs use $10 - 12$ M/A stages out of the 12 available. This does not leave any stages for post-inference processing in the ingress pipeline. However, it is possible to do additional processing in the egress pipeline where all the 12 stages are again accessible, as seen in Chapter 5.

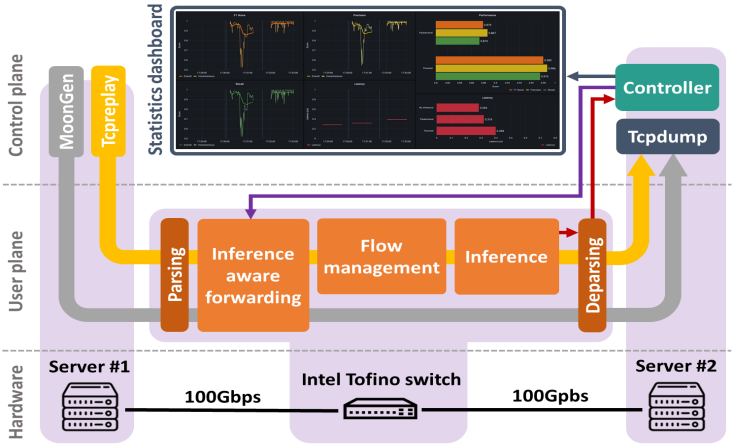


Figure 7.4: Demonstration workflow, and mapping of the logical components of the control and user planes into the testbed hardware.

To put all these numbers into perspective, it is worth noting that deploying the largest ETC model alongside the baseline P4 program for core L2/L3 switching functions, *i.e.*, `switch.p4` increases overall resource consumption by only 25% of what the baseline already requires. This sheds light on the memory efficiency of the ETC models. These results demonstrate how `Flowrest` can be adapted to classify encrypted traffic at line rate with high throughput and very low latency.

7.3. Live demonstrator

The demo setup in Figure 7.4 is used to showcase the in-switch FL inference that `Flowrest` enables [7, 11]. The setup consists of five logical components, which include (*i*) packet source, (*ii*) programmable switch, (*iii*) traffic sink, (*iv*) statistics dashboard, and (*v*) a controller that interacts with the switch, traffic sink, and the statistics dashboard. These components are mapped onto the hardware testbed, as illustrated in Figure 7.4, and detailed below.

Server #1 serves as a traffic source which generates and sends high-speed background traffic (illustrated by the grey arrow in Figure 7.4) to the switch using the MoonGen [118] packet generator. The packets that are targeted for inference (highlighted in yellow in Figure 7.4) are simultaneously injected into the switch by replaying historical pcap traces of the dataset using Tcpreplay [109]. Server #2 serves as a traffic sink which collects the traffic forwarded by the switch using a Tcpdump [117] instance. The collected traffic is analyzed by the controller.

The Intel Tofino switch is used to forward traffic from the source (Server #1) to the sink (Server #2), while performing inference on target flows. After parsing, the traffic is categorized as generic or target traffic in the inference-aware forwarding block, as shown in Figure 7.4. While generic traffic is forwarded normally, the target packets are either

classified directly in the case of the PL benchmark solutions, or first identified as belonging to specific flows in the case of `Flowrest`, then classified, and finally forwarded according to their class [2]. After each classification, a digest is sent to the controller with the class of the flow which it uses for table updates and statistics.

A Python-based controller program runs on Server #2. It plays multiple roles. First, it configures the switch during each experiment by injecting the compiled P4 code of the relevant model via the Barefoot Runtime Interface (BRI) of the Tofino switch. Second, it analyzes the traces collected by the traffic sink to calculate the end-to-end packet-processing latency. These latency estimates enable a verification of the estimated latencies that the P4 Insight tool provides. Third, it calculates the inference performance in terms of precision, recall, and F1-score by comparing the classification results with the ground truth. Finally, the controller feeds the statistics dashboard with data for display.

The Statistics Dashboard displays the performance of `Flowrest` in a dashboard connected to the Controller. The dashboard is organized into two main areas, as shown in Figure 7.4. In the first one (left), live results are shown while each evaluation is running. In the second one (right), results of the evaluations are summarized and compared.

7.4. Summary

This chapter presented a thorough experimental evaluation of the proposed `Flowrest` system for FL in-switch inference. Extensive experiments conducted with multiple use cases based on unencrypted and encrypted traffic datasets demonstrated the generalizability of the solution, and shed light on how it outperforms five representative works in various inference tasks. To the best of available knowledge, `Flowrest` is the first work to perform such a thorough evaluation against multiple existing solutions at PL and FL, in terms of classification accuracy and resource usage. The ability of the framework to accommodate any feasible model mapping has also been demonstrated. This establishes `Flowrest` as a viable tool that can provide FL functionality to any PL inference scheme, irrespective of whether it is tree-based, as long as the model is feasible in hardware.

Future work could explore improving the classification accuracy, experimenting with datasets featuring a much larger number of classes, and tackling use cases where traffic is entirely tunnelled, such that it is difficult to identify flows via their 5-tuples, in which case classifying entire sessions might be more appropriate. In addition, exploring the limits of FL inference and designing a more robust user-plane inference solution is another line of research that remains open and which is the subject of Chapter 8.

Part III

Joint inference

Packet-Level (PL) solutions for user-plane inference are simple to deploy and can enable inference on all packets going through the switch [3,4,10]. However, their simplicity means that they are unable to leverage important statistical Flow-Level (FL) statistics to enrich their model feature set and enhance classification accuracy.

On the other hand, F-L solutions *e.g.*, `Flowrest` [2, 5] have to observe at least a few packets of a flow in order to compute FL features and perform inference. In the process, they miss several packets which go through unclassified.

In this part of the thesis, the above limitations of PL and FL inference are tackled and a novel solution is proposed which offers the best of both worlds by enabling joint PL and FL inference in the user plane, using a single model.

8 Jewel: Joint packet-level and flow-level inference

A major dichotomy exists between existing works for Machine Learning (ML) inference in the user plane. This splits them into two large subsets, depending on whether they perform inference on individual packets [4,38–40,43,93,95,96], or on flows [2,41,42,44,91, 94,97,98]. Both approaches are illustrated in Figure 8.1. Packet movement is from left to right, with the first packet being the right-most on each row. Four rows of 8 packets each are shown. The first shows how features are extracted from the packets, while the remaining three respectively depict Packet Level (PL), Flow Level (FL), and joint PL and FL inference. Two vertical lines separate the flow of packets into three phases; before FL classification, during FL classification, and after FL classification.

In PL inference, individual packet header fields (*e.g.*, packet length or transport protocol) are used as features. PL models are stateless and very lean, as they only need to operate on one packet at a time as in the baseline forwarding tasks the switch is designed for. Thus, they cannot leverage FL features, and can pay a price in terms of accuracy as the problem complexity grows, as exemplified by the errors on the 6^{th} and 7^{th} packets in Figure 8.1. On the other hand, FL solutions implement fairly complex strategies to store stateful per-flow features (*e.g.*, Inter-Arrival Time (IAT) or maximum packet size, see Figure 8.1), which grant higher performance at the cost of added switch resource consumption.

An often disregarded trade-off between PL and FL models is that PL models, although less accurate, can classify *all* packets; whereas the more precise FL solutions are obliged to delay inference until the moment when FL features are reliable, and thus, *cannot classify early packets* in each flow [85], as also illustrated for the first 3 packets of the target flow on the F-L Inference row in Figure 8.1. While these early packets are not incorrectly classified by FL models, no action is taken on them either, and the impact is ultimately similar: for instance, in attack detection use cases, the first packets of a malicious flow could be let through as benign and may harm the system. The number of early flow packets ignored by FL models vary depending on the model and application use case from 2 to 50 in the existing literature. Yet, an analysis of the traffic in the

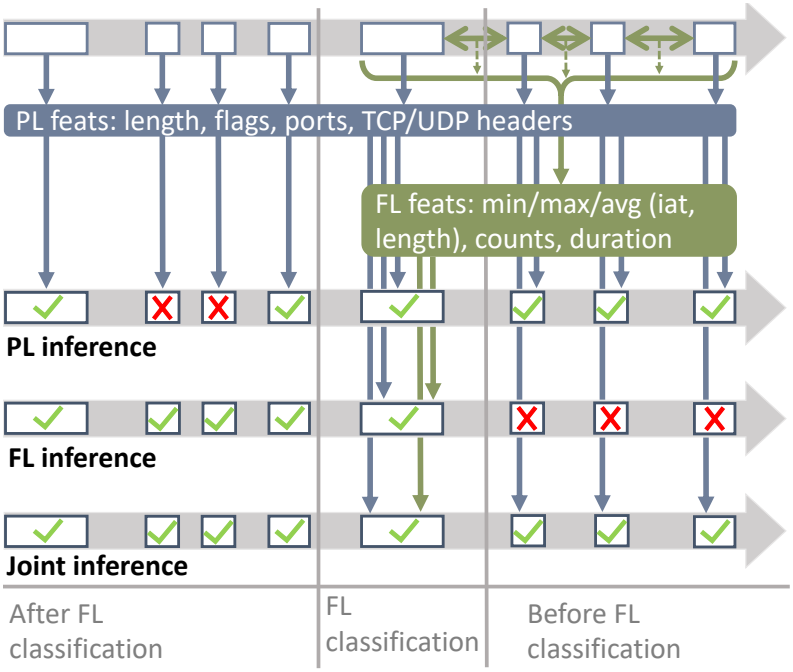


Figure 8.1: Illustration of packet-level (PL), flow-level (FL) and joint inference performed by in-switch ML models.

four measurement datasets considered for performance evaluation in this chapter reveals that: (*i*) the fraction of flows composed by less than 2 and 50 packets, which are entirely skipped by FL models, is up to 46.44% and within 47.51–99.87%, respectively; and (*ii*) for flows longer than 2 and 50, early packets that go unclassified by FL models account for up to 67.67% and 98.04% of the total flow length, respectively. Overall, these figures show how FL inference can be oblivious to a non-negligible portion of traffic in realistic use cases, highlighting a problem ignored to date.

This chapter closes the gap above by proposing an in-switch ML model that serves as a joint packet and flow-level classifier, *i.e.*, `Jewel` [8]. The solution enables PL inference on early packets, and shifts to more accurate FL classification as soon as FL features are ready. By doing so, `Jewel` provides the best of both worlds, as exemplified in Figure 8.1: early packets (packets 1–3) are correctly processed by the in-switch ML model, and at the same time higher accuracy is achieved at flow level (from packet 4).

By developing `Jewel`, the following contributions are made.

- **Fully joint PL–FL model design.** A straightforward approach to realize joint PL–FL operation is to deploy multiple ML models in the switch, dedicated to PL or FL inference and triggered on the appropriate packets. In fact, this is the approach adopted by the only solution for PL–FL inference proposed to date, *i.e.*, NetBeacon [101]. However, as shown in the evaluation conducted in this chapter, this strategy is very expensive in terms of switch resources, which in turn constrains the complexity of the PL and FL models, reducing their performance. Instead,

`Jewel` is proposed as an original ML model design that is *fully joint*, *i.e.*, can perform both PL and FL inference via a single Random Forest (RF), automatically switching between the two modes depending on the packet.

- **Hardware-native ML training.** A joint PL–FL solution naturally tends to be more resource-hungry than the PL or FL models, which calls for improved techniques for minimizing the memory footprint of the deployed ML model. The internal organization of the switch hardware is exploited to optimize feature representation and tree structures for fitting in scarce resources like Ternary Content-Addressable Memory (TCAM).

- **Experimental comparative evaluation.** `Jewel` is implemented as open-source[1] P4 software and encoded into a production-grade Intel Tofino switch in the hardware testbed presented in Chapter 3, alongside four benchmarks that represent the state of the art in PL, FL and joint inference. Experiments are conducted with four different use cases based on measurement data to demonstrate how `Jewel` consistently outperforms all existing solutions for in-switch ML in terms of classification accuracy, while keeping resource usage under control. In particular, the fully joint PL–FL design substantially enhances the steadiness of `Jewel`, which performs well in all settings, whereas benchmarks have much more erratic results across use cases. By averaging over all scenarios, `Jewel` yields 3.2% higher accuracy than the second-best model proposed to date.

8.1. Joint in-switch flow-packet classification

The design and operation of `Jewel` is presented by first providing an overview of the solution (Section 8.1.1), and then detailing the RF model for fully joint PL–FL inference (Section 8.1.2), a summary of the automated feature and model selection routines (Section 8.1.3), and the final implementation in hardware (Section 8.1.4).

8.1.1. Jewel in a nutshell

Figure 8.2 outlines the operation of `Jewel` from a high-level perspective, which is modelled after that of recent FL inference models for programmable switches [2, 41]. At step ①, the model is prepared and trained in the control plane via a software tool that explores all the possible combinations of hyper-parameters and features, takes into account the hardware constraints, optimizes the memory usage, and finally selects the model with the best F1-score. The model is then mapped at step ② into the switch architecture via P4, built and injected into the pipeline by the controller.

[1] The `Jewel` source code is available on GitHub at https://github.com/nds-group/Jewel.

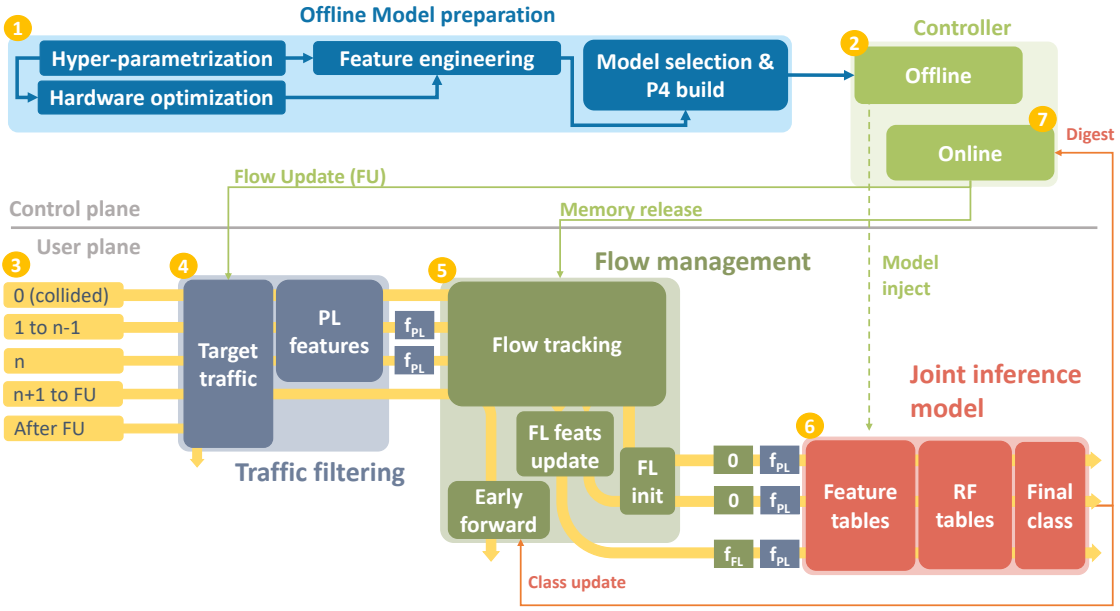


Figure 8.2: Overview of `Jewel` operation. Steps ①-⑦ denote the order of events.

The user plane implementation that performs the actual inference on incoming packets, depicted by ③, consists of three main elements. The traffic filtering, in step ④, identifies the target flows subject to inference (which may not represent the entirety of traffic) and extracts PL features from the relevant headers. The flow management, in ⑤, maintains a flow tracking register about the flows for which FL features are being collected, and stores and updates such FL features that are in preparation. Finally, the joint inference model, in ⑥, exploits PL and FL (if available) features for classification.

The yellow arrows in Figure 8.2 represent five different paths that incoming packets can follow in the switch pipeline, according to their order of arrival in the flow they belong to. Assuming that FL inference is triggered at packet number `n` in the flow, all early packets from 1 to `n-1` are processed for PL feature extraction, and go through flow management where they are used to update the FL features that are under preparation for the flow. Then, these packets go through inference using only the available PL features. Packet number `n` in the flow follows the same path as the earlier ones, but in addition, retrieves the final value of the FL features after having updated them. Packet `n` thus undergoes inference with the full set of PL and FL features. The result of this classification is particularly important, as it determines the class that will be assigned to all packets of the flow that arrive after `n`. As seen in Figure 8.2, packets from `n+1` are redirected by flow tracking to an early forwarding table where they are tagged with the class determined for the whole flow based on `n`.

Upon classification, a digest is sent to the controller with the information of the class of the flow. The controller then triggers two actions, in ⑦: (*i*) a Flow Update (FU) that inserts directly into the target traffic table the egress port of the target flow, based on the

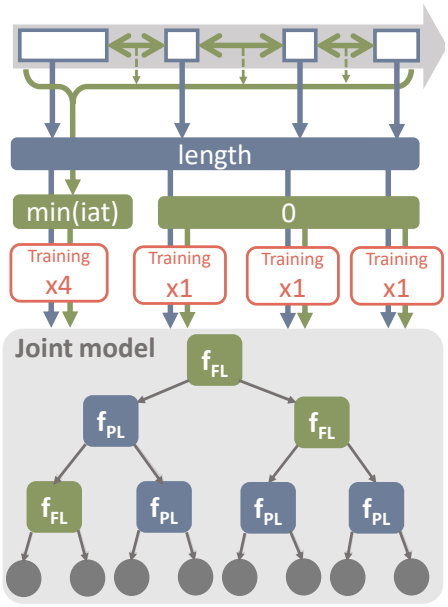


Figure 8.3: Joint PL-FL RF example.

class according to a predetermined policy (*e.g.*, malicious packets may be redirected to a honeynet for further analysis); and (*ii*) the release of the flow tracking registers occupied by the flow. Therefore all incoming packets of the flow arriving after the FU are forwarded according to the traffic table only, as also illustrated in the figure.

A peculiar case is that of packets tagged as `0` in Figure 8.2, which belong to flows that *collide* in the flow management step. The flow tracking register is typically implemented as a hash table with a finite size, meaning that two different flows may be hash to the same register index. In case of such a collision, the packet cannot initiate the FL operations, and `Jewel` simply proceeds to inference using PL features only. It is worth noting that, instead, pure FL solutions cannot classify all packets in collided flows unless other mitigation strategies are adopted.

8.1.2. Fully joint PL-FL model design

At the core of `Jewel` is an original ML model design that can perform both PL and FL inference via a single RF, which automatically adapts the mode of operation depending on the position of the packet within the flow. Thus, the block labelled ⑥ in Figure 8.2 is realized with a single, monolithic RF. To this end, an RF model is devised that receives as input a complete PL and FL feature set, and operates according to a simple but effective idea illustrated with a toy example in Figure 8.3. As in Figure 8.1, packet movement is from left to right. Thus, the 1st packet is the right-most packet and the 4th packet if the left-most one. The figure is explained next.

- During the training phase, for packets from `1` to `n-1`, the feature set is composed of the actual PL features of the packet (*e.g.*, the packet length in the

sample), plus *default* constant values never attained in reality for all FL features (*e.g.*, a null inter-arrival time in the example).

- When training gets to packet `n`, for which FL features become available, the feature set is switched to actual values of both PL and FL features (*e.g.*, in the example, the packet length and the actual minimum inter-arrival time measured on the first four packets of the flow).

- An important observation is that *weights* are assigned to individual samples (*i.e.*, packets) to account for the fact that their inference outcome does not have the same impact on the final result. Specifically, the first `n-1` packets are each given a weight 1, as their correct classification means that one packet will be accurately tagged by the ML model during the actual inference phase. Instead, packet `n` is attributed a higher weight, since its classification affects all the following packets in the flow, which, as explained in Section 8.1.1, inherit the FL class. Specifically, the higher weight is set to `n` (*e.g.*, 4 in the toy example), which allows balancing the attention of the model between its PL and FL inferences[2].

Through the strategy adopted above, the RF model is taught to rely on PL features only when FL features are assigned the default values. Indeed, such default values are constant for all packets and completely uncorrelated to the variable to predict, such as the category or adversarial nature of the packet. In other words, the RF learns to discard default values for FL features. Of course, when actual FL features are available, the RF also learns to use them to improve the classification accuracy. The operation during inference is then straightforward, and is depicted at the input of block ⑥ in Figure 8.2: whenever the FL features are not available, *i.e.*, for packets `1` to `n-1` or in collided flows, default FL feature values are instead used.

The `Jewel` approach is referred to as *fully joint*, as it actually uses a single model, in contrast to NetBeacon [101], which is the only solution for joint PL–FL inference in the literature and which relies on separate models for PL and FL classification. As it will be shown in Section 8.3, the fully joint design of `Jewel` allows for a more compact and yet accurate model, which achieves substantial gains in both accuracy and resource usage over NetBeacon.

8.1.3. Hardware-tailored feature and model selection

Another key element of the design of `Jewel` is the original execution of the offline model preparation phase, marked as ① in Figure 8.2. An automated model analysis tool is developed that explores multiple combinations of (*i*) features and (*ii*) hyper-parameters

[2]Other weight configurations were also tested, *e.g.*, the total number of packets in the flow. However, they exceedingly downplayed the importance of PL inference, leading to a reduced overall performance.

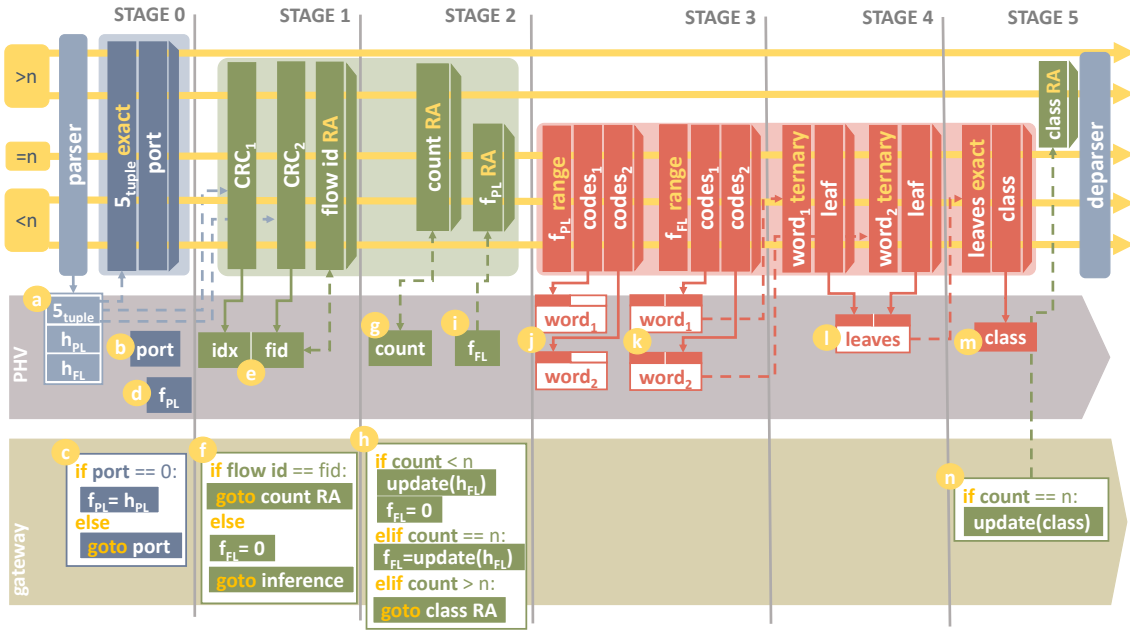


Figure 8.4: Detail of the mapping of `Jewel` into the PISA architecture.

like the maximum tree depth, maximum number of leaves, number of trees and value of `n`, *i.e.*, the rank of the packet for which the FL inference is triggered. The pipeline selects such variables while considering *by design* the inherent constraints of the underlying hardware, ensuring that the selected model can be ultimately deployed in the switch using the mapping methodology detailed in Section 8.1.4. The details of the automated model analysis tool are omitted in this thesis since its development was led by another researcher. Full details can be found in the `Jewel` paper [8].

8.1.4. In-switch implementation and operation

The in-switch implementation of `Jewel` was led by a collaborating researcher, building on `Flowrest` and extending it to enable joint PL and FL inference. Details of the implementation are included in this thesis for the sake of completeness of the description of the solution. Any missing details can be found in the paper [8].

Figure 8.4 shows the mapping of `Jewel` onto a Protocol Independent Switch Architecture (PISA) pipeline. For simplicity, the figure depicts a toy example of a model with two features, one PL and one FL, and an RF with two trees. The top layer in the figure depicts the sequence of Match & Action Table (MAT) and Register Action (RA) entities traversed by the packets in the different stages. The packets entering the switch are divided into three groups, highlighted in yellow. From bottom to top, the packets before `n`, packet `n`, and those after `n` are shown. From each of those groups of packets, five arrows depart and traverse the pipeline from left to right. Such arrows identify the different paths that can be followed, consistent with the operation outlined in Figure 8.2.

In the middle layer, the Packet Header Vector (PHV) carries the headers extracted from the packets and the metadata variables generated at different points of the pipeline. The interactions between the top layer and the PHV are also portrayed as arrows. Dashed arrows refer to matching, in the case of MATs, and comparison or updates in the case of RAs. Solid arrows indicate an assignment to metadata. The bottom layer shows the Gateway where the if/else conditions applied to the data of the PHV reside. The interactions between the PHV and the Gateway are highlighted with encircled letters, from ⓐ to ⓜ.

When a packet enters the switch pipeline, three groups of relevant header fields are extracted: the 5-tuple characterizing a flow, composed of source and destination IP addresses, source and destination port and protocol; the header fields $\mathtt{h_{PL}}$ used as the PL feature; the header fields $\mathtt{h_{FL}}$ employed to calculate the FL feature. The 5-tuple is matched against the flow table to determine whether the flow is a target for inference, and if there is a match, the egress port is retrieved. A value of the port different from zero means that the flow has already been classified and the packet is forwarded with no further action. Otherwise, the value of $\mathtt{h_{PL}}$ is assigned to the PL metadata $\mathtt{f_{PL}}$.

In the following stage, the 5-tuple is used to compute two Cyclic Redundancy Check (CRC) hashes of 16 and 32 bits, to calculate the flow register index `idx` and the flow identifier `fid`, respectively. The index is used to access the entries of the RAs allocated to the flow. Three RAs' registers store respectively, the flow id, the count of the packets, and the FL features $\mathtt{f_{FL}}$ updated to the last packet seen. If the `fid` is the same as the flow identifier in the RA (or if the latter is empty) the packet is sent to the count RA. Otherwise, a flow collision is detected, $\mathtt{f_{FL}}$ is set to `0` and the packet is advanced to the inference stage for PL classification.

In the next stage, the count RA is incremented and three different cases are identified according to its updated value. If it is less than `n`, the $\mathtt{f_{FL}}$ RA is updated using $\mathtt{h_{FL}}$ and $\mathtt{f_{FL}}$ is set to 0, *i.e.*, the packet is marked for PL inference. If the count is `n`, the $\mathtt{f_{FL}}$ RA is updated and its value is assigned to the $\mathtt{f_{FL}}$ metadata, *i.e.*, the packet is prepared for FL inference. If the count is greater than `n`, the packet is sent to the class RA, where it is forwarded based on its already determined class.

Starting from the next stage, the joint model is encoded in a sequence of MATs. The RF mapping presented in Chapter 3 and used throughout this thesis is also employed in `Jewel`. In the toy example in Figure 8.4, one MAT per feature and one MAT per tree are employed, such that two feature MATs and two tree MATs are represented. Each feature MAT encodes the split decisions (*i.e.*, conditions on whether one feature is higher or lower than a trained threshold) of both trees. The splits are encoded in the form of ranges, with the first range covering values from 0 to the smallest threshold present in the splits of the target feature, and each subsequent range encompassing the values up to the following threshold. Each range match can then be associated to an action, *i.e.*, a

code, for each tree, which encodes the outcomes of all the splits where the feature occurs. For instance, in Figure 8.4, when $\texttt{f}_{\texttt{PL}}$ is matched against the PL feature MAT, $\texttt{code}_1$ and $\texttt{code}_2$ are retrieved to partially fill $\texttt{word}_1$ and $\texttt{word}_2$, respectively. The same process is repeated with $\texttt{f}_{\texttt{FL}}$ against the FL feature MAT, filling completely $\texttt{word}_1$ and $\texttt{word}_2$. The metadata variables $\texttt{word}_1$ and $\texttt{word}_2$ encode the outcomes of all the splits of the first and the second tree, respectively.

In the next stage, the metadata variables $\texttt{word}_1$ and $\texttt{word}_2$ are matched against their respective tree MATs. Each tree MAT encodes all the different combinations of split outcomes leading to a different leaf of the tree. Such combinations are masked using ternary matches according to the specific subset of bits, *i.e.*, splits, that lead to a specific leaf. In Figure 8.4, $\texttt{word}_1$ and $\texttt{word}_2$ are matched against their respective tree MATs and fill a `leaves` metadata that contains the classes predicted by each of the two trees.

In the final stage, the `leaves` metadata is exactly matched against a final table encoding all combinations of possible predictions by the two trees. The class associated with each match is the most voted one. All the cases with no majority are instead encoded as MAT entries that assign as the final class that of the tree with the highest accuracy.

8.2. Experimental setup

`Jewel` is implemented in an experimental platform with industry-grade programmable switches presented in Chapter 3, along with four state-of-the-art benchmarks described in Section 8.2.1. Then experiments are run comprehensively with four tasks based on real-world traffic (Section 8.2.2), collecting relevant performance metrics (Section 8.2.3).

8.2.1. Benchmarks

`Jewel` is compared against four recent solutions for in-switch ML that are representative of the state of the art across Decision Tree (DT) and RF models, as well as PL and FL inference strategies. The choice of benchmarks is based on the attained accuracy and model scalability, and this leads to two PL models, *i.e.*, Planter [43] and Mousika [93], one FL model, *i.e.*, Flowrest [2], and the only PL+FL model available in the literature, *i.e.*, NetBeacon [101]. The benchmarks are described in Chapters 2 and 7.

It is important to highlight that none of these models include a systematic model selection routine like that employed by `Jewel`. As it will be later shown in the experimental evaluation, the identification of the best model is paramount to ensure the optimal operation of the ML model. Therefore, for the sake of fairness, the proposed model selection procedure described in Section 8.1.3 is adapted to the benchmarks so that it can inform the choice of (*i*) the best PL-only model, which is then employed by both Planter and Mousika, and (*ii*) the best FL-only model, used by Flowrest. As far as NetBeacon is concerned, both (*iii*) the best PL model with one tree is to implement the DT that

NetBeacon adopts to classify individual packets, and (*iv*) the best FL model with one tree is used to implement its per-flow DT classifiers at different packet numbers.

The overall aim is to put the benchmarks in the best position possible to compete with `Jewel`. Yet, that also means that the performance reported in Section 8.3 for Planter, Mousika, Flowrest, and NetBeacon represents best scenarios that would be hard to identify without the help of the novel model selection routine introduced in `Jewel`.

8.2.2. Use cases

A range of measurement-based classification tasks, which build on open datasets are selected. They enable an assessment of the robustness of the different solutions across heterogenous scenarios. The four datasets used were also employed in the evaluation of Flowrest [6] and include: the UNIBS Internet traces [79, 162] which is a service classification task with 8 classes; the UNSW-IoT traces [78] which is a device identification use case with 26 classes; the IoT-23 dataset [119] which is a bot classification use case with 14 classes; and the ToN-IoT dataset [120] for attack classification with 10 classes. The datasets are described in Chapter 3.

8.2.3. Metrics

The classification accuracy of `Jewel` and the benchmarks is evaluated using the widely adopted precision, recall, and F1-score metrics; with the micro, macro, and weighted averages considered. These metrics are presented in Chapter 3.

To ensure a fair comparison between the benchmarks, all the metrics are computed on a per-packet basis so that all solutions are judged on the same number of samples. This is straightforward in the PL models, where packets are the actual unit of operation. In the case of the FL or joint models, per-packet scores are obtained by extending the result of the classification of a flow to all packets following the one that triggered the FL classification (*e.g.*, after packet `n` in `Jewel`). Also, all early flow packets that go unclassified by FL models are considered not to be correctly classified, as the inference is completely oblivious to them.

8.3. Experimental results

`Jewel` is evaluated in terms of classification accuracy and consumption of switch resources, and compared to the four benchmark solutions described above.

8.3.1. Classification accuracy

Tests with real-world programmable switches highlight the superiority of `Jewel` across all scenarios in terms of classification accuracy. Table 8.1 compares the different

	Mousika	Planter	Flowrest	Netbeacon	Jewel
UNIBS	90.351%	91.560%	96.398%	94.570%	**98.354%**
UNSW	82.003%	79.853%	80.691%	78.594%	**87.317%**
ToN IoT	27.554%	70.496%	73.461%	70.063%	**75.703%**
IoT-23	86.054%	88.147%	82.857%	86.076%	**91.314%**

Table 8.1: Average of F1-score metrics across models and use cases.

models in terms of the average of their micro, macro and weighted F1-score. `Jewel` is invariably the best, with gains in the range 2.0–5.3% with respect to the second-ranked model, and typically of 10% or more over the worst benchmark. These are not negligible improvements, considering that the competitors are all amongst the best solutions currently available, which `Jewel` outperforms across very diverse use cases.

The full results from all the experiments are presented in Table 8.2. In fact, *consistency* in good performance is a remarkable characteristic of the proposed model. The second-best model mentioned above changes for each use case: Flowrest yields better accuracy than NetBeacon, Planter and Mousika in the tasks proposed in the UNIBS and ToN-IoT datasets, but Mousika and Planter take that role across the other two use scenarios, with second-best accuracy in UNSW and IoT-23, respectively. Each state-of-the-art model thus has properties that allow it to surpass other benchmarks in a specific scenario, yet penalize it in others. This is not the case for `Jewel`.

Comparing `Jewel` directly to NetBeacon *i.e.*, the only other PL+FL solution, `Jewel` always performs better, with absolute gains in the 3.8–8.7% range. This is attributed to two reasons. First, as NetBeacon's FL models are all single-tree RFs, their accuracy is limited compared to `Jewel`'s multiple-tree RF. Second, in NetBeacon, the PL model is used each time FL inference cannot be performed. This includes cases where flows experience hash collisions, as well as flows that are too short to be considered for FL inference. Thus, many packets might only be classified at PL, hence, an overall lower accuracy than `Jewel`, which instead prioritizes FL inference.

Figure 8.5 takes a deeper look at the unwavering quality of the proposed model across diverse settings. It breaks down the result above across micro, macro and weighted F1-score. To better appreciate the advantage of `Jewel`, the plot shows the difference in performance between each model and the model that performs the best. It is apparent how `Jewel` steadily achieves the highest average classification accuracy across all F1-score metrics. In other words, `Jewel` invariably leads a range of state-of-the-art in-switch inference models when considering the classification quality at both the level of individual packets and that of whole flows. The result is impressive, considering that optimizing different F1-scores is a hard task: this is evidenced in Figure 8.5 by the fact that all benchmarks exhibit substantial loss of accuracy for one F1-score metric or another, in one or multiple use cases. Similar trends are visible across the other metrics *i.e.*, precision and recall.

Dataset	Average	Metric	Mousika	Planter	Flowrest	NetBeacon	Jewel
UNIBS	Micro	Precision	92.565%	92.387%	**99.111%**	95.571%	98.528%
		Recall	92.565%	92.387%	96.968%	95.571%	**98.528%**
		F1-Score	92.565%	92.387%	98.028%	95.571%	**98.528%**
	Macro	Precision	90.471%	90.918%	96.329%	96.680%	**98.749%**
		Recall	87.265%	89.972%	90.294%	91.869%	**97.383%**
		F1-Score	87.834%	90.316%	93.183%	93.196%	**98.023%**
	Weighted	Precision	90.016%	91.718%	**99.084%**	95.568%	98.531%
		Recall	92.565%	92.387%	96.968%	95.571%	**98.528%**
		F1-Score	90.653%	91.978%	97.984%	94.943%	**98.512%**
UNSW	Micro	Precision	84.653%	85.834%	88.883%	85.744%	**91.441%**
		Recall	84.644%	85.834%	82.740%	85.744%	**91.441%**
		F1-Score	84.649%	85.834%	85.701%	85.744%	**91.441%**
	Macro	Precision	76.208%	68.834%	78.403%	67.899%	**81.435%**
		Recall	**85.531%**	73.475%	69.415%	70.303%	80.634%
		F1-Score	77.030%	67.844%	70.550%	63.537%	**78.584%**
	Weighted	Precision	88.453%	87.697%	91.086%	89.473%	**92.502%**
		Recall	84.644%	85.834%	82.740%	85.744%	**91.441%**
		F1-Score	84.330%	85.882%	85.823%	86.502%	**91.592%**
ToN-IoT	Micro	Precision	27.780%	79.104%	**89.086%**	79.116%	85.347%
		Recall	27.780%	79.104%	78.961%	79.116%	**85.347%**
		F1-Score	27.780%	79.104%	83.719%	79.116%	**85.347%**
	Macro	Precision	23.556%	58.889%	**63.281%**	59.049%	63.204%
		Recall	40.260%	52.700%	49.583%	51.991%	**54.386%**
		F1-Score	21.041%	51.304%	52.691%	50.347%	**54.571%**
	Weighted	Precision	56.235%	84.872%	**91.136%**	85.254%	90.396%
		Recall	27.780%	79.104%	78.961%	79.116%	**85.347%**
		F1-Score	33.840%	81.079%	83.973%	80.725%	**87.192%**
IoT-23	Micro	Precision	92.136%	92.587%	94.578%	92.512%	**95.635%**
		Recall	92.136%	92.587%	87.046%	92.512%	**95.635%**
		F1-Score	92.136%	92.587%	90.655%	92.512%	**95.635%**
	Macro	Precision	73.098%	87.863%	82.098%	79.305%	**92.580%**
		Recall	73.098%	77.996%	65.859%	74.333%	**81.896%**
		F1-Score	74.557%	79.300%	67.818%	72.935%	**83.045%**
	Weighted	Precision	93.807%	95.083%	97.604%	94.744%	**95.780%**
		Recall	92.136%	92.587%	87.046%	92.512%	**95.635%**
		F1-Score	91.469%	92.553%	90.097%	92.780%	**95.261%**

Table 8.2: Classification performance of Jewel and the benchmarks. The best score on each row is in bold.

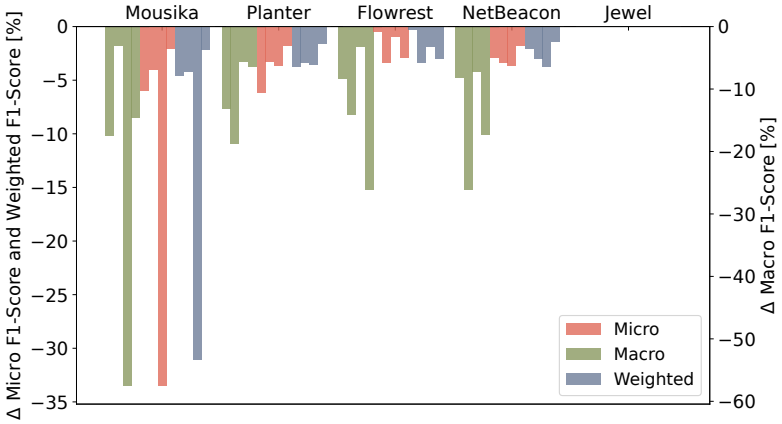


Figure 8.5: Accuracy loss of the benchmarks across the three F1-score metrics with respect to the most accurate model in each use case

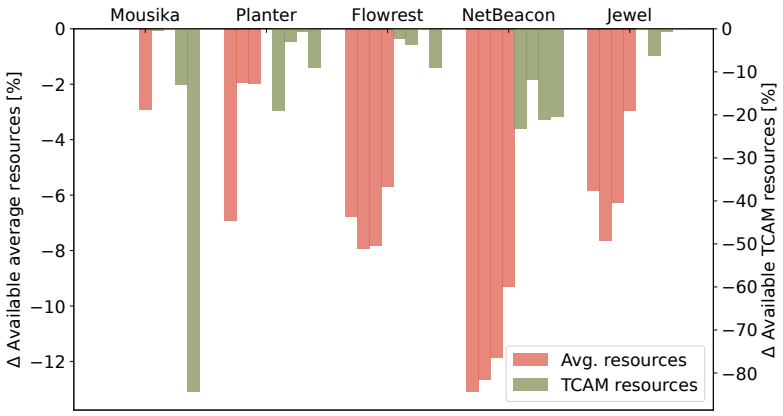


Figure 8.6: Reduction of available total and TCAM resources with respect to the most parsimonious model in each use case.

Overall, `Jewel` outshines the benchmarks, except for a few cases where Flowrest takes the lead in terms of precision, and one case where Mousika dominates in terms of macro recall. This further re-affirms `Jewel`'s consistency in achieving high classification performances, irrespective of the metric considered, thereby establishing it as the new standard for line-rate in-switch inference with RF models.

8.3.2. Resource usage

Finally, the consumption of switch resources is examined, as it is an important parameter when working with resource-constrained user-plane equipment. Figure 8.6 shows the reduction of available resources with respect to the most frugal model, in terms of both total available switch resources and the expensive TCAM.

As expected, PL models generally have a smaller resource footprint, especially when considering the total available resources. Yet, `Jewel` (*i*) is on par with an FL-only model like Flowrest, despite the added PL functionalities, and (*ii*) is much more resource-efficient than its direct PL+FL competitor, *i.e.*, NetBeacon. The lower resource consumption relative to NetBeacon is not surprising. This is because while NetBeacon deploys multiple PL and FL models for the same inference task, `Jewel` instead trains and deploys a single model, consuming less resources. When looking at TCAM, *i.e.*, the memory optimization target during model selection, `Jewel` is often the most effective model, saving resources even over PL models. This demonstrates the resource-efficiency of the proposed solution and the performance of the model selection process which gave rise to these models.

Overall, the comparative evaluation highlights how `Jewel` offers an excellent trade-off between the best accuracy and a resource-efficient in-switch implementation, largely outperforming its direct competitor, *i.e.*, NetBeacon. This sets `Jewel` as the new state of the art for in-switch ML inference with RFs.

8.4. Summary

This chapter proposed `Jewel`, an original model for joint PL and FL in-switch inference, based on a single resource-efficient RF model, trained to address both types of classification. Extensive experiments conducted in a real-world testbed with production-grade equipment prove the high accuracy, strong consistency, and resource efficiency of `Jewel` across various use cases. Experimental results shed light on how `Jewel` outperforms four leading solutions for in-switch inference with RF models in terms of classification accuracy and resource efficiency, establishing it as a new standard.

While `Jewel` is memory-efficient compared to the benchmark solutions considered, its consumption of key resources like TCAM is not always negligible, going up to 15% in one of the use cases tested. As such, there is room for improvement in the design of more resource-conservative techniques for mapping models to switch pipelines which will enhance the memory efficiency of `Jewel`. Another possibility to reduce memory consumption would be to deploy `Jewel` in distributed scenarios, where the inference load is shared across different nodes in the network. For instance, one switch could be responsible for running the feature extraction and flow management parts of the pipeline, while another one runs the inference model, and sends the result to a third switch which then executes post-classification actions. This will minimize resource usage on each switch.

Finally, for most use cases examined in this chapter, inference could be performed on flows after observing as few as 2–4 packets. This means the PL model generally infers on very few packets, directly prioritizing FL inference. Future work could seek out use cases where flows are significantly longer in general, such that many more packets are required to infer the flow. This will enable testing the limits of joint inference and possibly improving the model design process.

9 Conclusions and perspectives

The growing need for network automation sparked by the increased complexity of networked systems in recent years has led to several efforts to implement intelligent applications in networks. User-plane Machine Learning (ML) inference contributes to those efforts by offloading models to the user-plane to enable inference at line rate while battling with the strict constraints of programmable user-plane environments. Prior work has progressed in embedding Decision Tree (DT) and Random Forest (RF) models into programmable switches for line-rate inference. Yet, their practicality, scalability, generalizability and performance on difficult use cases are limited.

9.1. Conclusions

In this thesis, shortcomings of prior work have been addressed and steps made towards a deeper integration of ML in the user plane by proposing solutions for embedding models into the user plane for Packet Level (PL), Flow Level (FL) and joint PL+FL inference.

In Chapter 2, the road to user-plane inference was traced from the inception of Software Defined Networking (SDN), through the advent of programmable user-plane equipment, to recent efforts to embed ML into the user plane. The focus was then narrowed down to in-switch ML with DT and RF models by performing an exhaustive review of existing works, discussing their strengths and exposing their limitations which have been addressed in this thesis. In Chapter 3, the experimental methodology common to all the proposed solutions was outlined, detailing how models are trained and implemented in programmable switches, followed by a description of the hardware testbed, metrics, and datasets used for evaluation.

The core contributions of the thesis have been presented in five chapters. Chapter 4 presented a practical application of user-plane PL inference for rapid cyberattack detection in Smart Grid (SG) systems which runs at line rate and detects various attacks up to 17,000 times faster than its control-plane counterparts. This represents a significant advancement of the state of the art. In Chapter 5, a novel hierarchical ML inference model

dubbed `Henna` was proposed to address scalability issues in flat PL classification models, which it outperforms in relatively complex tasks.

The next two chapters presented `Flowrest` (Chapter 6), and a thorough evaluation of its abilities to classify unencrypted and encrypted traffic (Chapter 7). Through the proposed design and evaluation, `Flowrest` has been established as the new standard for FL in-switch inference which outperforms existing solutions on a wide range of inference tasks while keeping resource consumption in check. `Flowrest` has also been shown to enable the easy conversion of any feasible PL solution to FL and to be generalizable to many inference tasks.

Finally, `Jewel` was proposed in Chapter 8 as a resource-efficient model for joint PL and FL inference in programmable switches. `Jewel` has been shown to achieve high classification accuracy by performing PL inference on early flow packets and then prioritizing FL inference when FL features are deemed reliable. By eliminating the need to deploy multiple PL and FL models in the switch to achieve joint inference, `Jewel`'s consumption of switch resources is modest, improving its scalability.

The solutions proposed in this thesis have significantly advanced the state of the art in user-plane inference and made steps towards a more seamless integration of ML models into commercial programmable user-plane equipment for line-rate inference at high throughput and with ultra-low latency.

9.2. Perspectives

The topic of user-plane inference in programmable networks continues to draw much attention. Hence, multiple directions for future research are envisaged in the immediate future and the long term.

9.2.1. Short-term perspectives

First, most of the proposed solutions to date including those outlined in this thesis have mainly focused on enabling inference in the user-plane. Future work could build on the recorded successes to efficiently integrate these solutions into networks to tackle real management problems *e.g.*, routing optimization, real-time anomaly detection, and QoS management. Engagement with industry network operators could facilitate efforts in that direction by gaining access to test production networks and more realistic use cases.

A second line of research would be the diversification of the inference targets and models used. This thesis focused on DT and RF models in programmable switches due to switches' multiple high-throughput ports and ubiquitous presence in the network, and the simplicity of tree-based models. While these models excelled in the use cases evaluated in this thesis, it would be interesting to explore the mapping of more model types to user planes so that there is a wider sample space of choices when picking a model for an

inference task. Also, FPGAs and SmartNICs are becoming ever more present in the user plane and exploring them for inference either individually or working in collegiality with switches is another possible research direction.

Another important aspect is the evaluation of the stateful FL solutions at Gbps speeds. This will enable an assessment of their ability to support inference at such high data rates and subsequently design alternative approaches to state storage and management in the user plane if these solutions fail to scale in such scenarios. There is recent work on millions of state insertions in ASIC switches which could be a starting point for such endeavours.

Lastly, as use cases become more complex, model complexity often grows, leading to the consumption of more switch resources. While most of the solutions proposed in this thesis are shown to consume small amounts of switch resources, their ability to cohabit with other key switch functions will depend on their complexity. Hence, further reducing the memory footprint of the user-plane inference solutions is an important research direction.

Closely related to this is the concept of distributed inference by which a large ML model is split into different parts that could each be deployed on different user-plane equipment. This approach goes beyond the hierarchical inference scheme proposed in this thesis and can enable the distribution of any task even if there are no obvious hierarchies between the classes. Determining how to split the models, how the different model parts will communicate, and how they will be coordinated are all questions that must be answered to make such a distribution realistic.

9.2.2. Long-term perspectives

Beyond the immediate future research directions described above, user-plane inference is expected to continue contributing to the transformation of the programmable network landscape. Thus, there is a lot of interest in developing new programmable user-plane targets. Although Intel announced in early 2023 that they would stop the development of the next-generation Intel Tofino Intelligent Fabric Processor (IFP) products, they continue to market the current products and have also proposed Infrastructure Processing Units (IPUs) based on either ASICs or FPGAs. More recently, they have introduced an Intel P4 Suite for FPGAs which will facilitate integrating FPGAs in programmable user-plane ecosystems. Also, many new vendors, *e.g.*, Pensando, Netronome, and NVIDIA are continuously developing new SmartNICs and Data Processing Units (DPUs) which will further expand the availability of programmable user planes and foster more applications.

In the early 2030s, the first modern 6G networks are expected to be rolled out. Work on 6G standardization is in its early stages but one thing is certain; 6G networks are envisioned to incorporate intelligence by design. Such intelligence would cut across the network stack and enable automation. ML will play an important role in enabling intelligence by design and in the user plane, ML inference will transform network data

transport by enabling line rate decision-making on network traffic as it is forwarded. Another approach would be to have a logical intelligence plane that communicates with all network planes and handles all the intelligent functions of the network. Such a plane would leverage and coordinate user-plane inference as part of its functions and enable native network intelligence.

The ultimate intelligent user plane will enable enhanced network security, through ultra-low latency detection and mitigation of attacks on the network; real-time network optimization, through the adaptation of routing and forwarding rules based on the intelligent user-plane model outputs; and personalized network experiences resulting from pervasive AI model deployments in the user plane. Such possibilities will move us closer to the vision of self-driving networks, where human-in-the-loop approaches to network management are rendered obsolete. The ideas presented in this thesis have contributed to laying the foundation for these exciting prospects.

Several long-term research directions could also emanate from this thesis and transcend user-plane inference. The first perspective will be to enable more complex user-plane applications by employing FPGAs and SmartNICs that are less constrained in terms of available resources. Notably, deep learning models running on FPGAs or SmartNICs will be employed for applications like network tomography which are more difficult to handle using tree-based models. A more radical approach to enabling more complex applications, albeit with reduced performance, will be to employ recent technologies like eBPF to deploy applications for network security, observability and management in the Linux kernel space. Many large corporations, *e.g.*, Google, Cloudflare, Android, and Netflix are using eBPF for security, network monitoring and security, and even for packet processing. eBPF thus appears to have a huge potential which will be tapped to enable more in-network applications that are more amenable to production networks.

Another long-term and more ambitious perspective is the design of a smart transport plane for 6G, through a seamless integration of ML into the 6G transport layer. The smart transport plane will boost 6G by lowering application latency through task offload, enhancing security through in-network attack detection and mitigation applications, facilitating the adoption of ML-assisted and ML-enabled automation by introducing more explainable ML models, contributing to the development of standards for smart 6G transport, and improving network performance using real-time analytics for traffic routing optimization. These ideas will contribute to defining what exactly 6G would be.

In summary, user-plane inference will continue to inspire new research avenues in the programmable networking space, which will go beyond the scope of this thesis and the perspectives described above. Remaining up-to-date with the state-of-the-art and continuously contributing to advancing research in the area, will be the way forward, building on this thesis in the long term.

References

[1] A. T.-J. Akem and M. Fiore, "Towards data-driven management of mobile networks through user plane inference," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2024, pp. 1–4. [Online]. Available: https://dspace.networks.imdea.org/handle/20.500.12761/1801

[2] A. T.-J. Akem, M. Gucciardo, and M. Fiore, "Flowrest: Practical flow-level inference in programmable switches with random forests," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10. [Online]. Available: https://doi.org/10.1109/INFOCOM53939.2023.10229100

[3] ——, "Ultra-low latency user-plane cyberattack detection in sdn-based smart grids," in *Proceedings of the 15th ACM International Conference on Future and Sustainable Energy Systems*, ser. e-Energy '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 676â682. [Online]. Available: https://doi.org/10.1145/3632775.3661995

[4] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, "Henna: hierarchical machine learning inference in programmable switches," in *Proceedings of the 1st International Workshop on Native Network Intelligence*. New York, NY, USA: Association for Computing Machinery, 2022, pp. 1–7. [Online]. Available: https://doi.org/10.1145/3565009.3569520

[5] A. T.-J. Akem, G. Fraysse, and M. Fiore, "Encrypted traffic classification at line rate in programmable switches with machine learning," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2024, pp. 1–9. [Online]. Available: https://hdl.handle.net/20.500.12761/1791

[6] A. T.-J. Akem, B. Bütün, M. Gucciardo, and M. Fiore, "Practical and general-purpose flow-level inference with random forests in programmable switches," *IEEE/ACM Transactions on Networking*, pp. 1–16, 2024.

[7] ——, "Showcasing in-switch machine learning inference," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 299–301. [Online]. Available: https://doi.org/10.1109/NetSoft57336.2023.10175464

[8] ——, "Jewel: Resource-efficient joint packet and flow level inference in programmable switches," in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, 2024. [Online]. Available: https://hdl.handle.net/20.500.12761/1777

[9] M. Gucciardo, B. Bütün, A. T.-J. Akem, and M. Fiore, "Evaluating the impact of flow length on the performance of in-switch inference solutions," in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2024, pp. 1–6. [Online]. Available: https://hdl.handle.net/20.500.12761/1800

[10] B. Bütün, A. T.-J. Akem, M. Gucciardo, and M. Fiore, "Fast detection of cyberattacks on the metaverse through user-plane inference," in *2023 IEEE International Conference on Metaverse Computing, Networking and Applications (MetaCom)*, 2023, pp. 350–354. [Online]. Available: https://doi.org/10.1109/MetaCom57706.2023.00067

[11] M. Gucciardo, A. T.-J. Akem, B. Bütün, and M. Fiore, "Demonstrating flow-level in-switch inference," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2023, pp. 1–2. [Online]. Available: https://doi.org/10.1109/INFOCOMWKSHPS57453.2023.10225967

[12] N. Feamster and J. Rexford, "Why (and how) networks should run themselves," 2017. [Online]. Available: https://arxiv.org/abs/1710.11583

[13] P. Kalmbach, J. Zerwas, P. Babarczi, A. Blenk, W. Kellerer, and S. Schmid, "Empowering self-driving networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*. NY, USA: ACM, 2018, pp. 8–14. [Online]. Available: https://doi.org/10.1145/3229584.3229587

[14] European Telecommunications Standards Institute (ETSI), "Zero-touch network and Service Management (ZSM); Proof of Concept Framework," ETSI GS ZSM 006 V1.2.1, Feb. 2022. [Online]. Available: https://www.etsi.org/deliver/etsi_gs/ZSM/001_099/006/01.02.01_60/gs_ZSM006v010201p.pdf

[15] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, "Aztec: Anticipatory capacity allocation for zero-touch network slicing," in *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, 2020. [Online]. Available: https://doi.org/10.1109/INFOCOM41043.2020.9155299

[16] C. Kilinc, C. Sun, and M. K. Marina, "5G development: Automation and the role of artificial intelligence," *Wiley 5G Ref*, pp. 1–29, 5 2020. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/9781119471509.w5GRef128

[17] R. Amin, E. Rojas, A. Aqdus, S. Ramzan, D. Casillas-Perez, and J. M. Arco, "A survey on machine learning techniques for routing optimization in SDN," *IEEE Access*, vol. 9, pp. 104 582–104 611, 2021.

[18] Y. Yoo, G. Yang, C. Shin, J. Lee, and C. Yoo, "Machine learning-based prediction models for control traffic in SDN systems," *IEEE Transactions on Services Computing*, vol. 16, no. 06, pp. 4389–4403, nov 2023.

[19] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring control plane latency in SDN-enabled switches," ser. SOSR '15. ACM, 2015.

[20] The 5G Infrastructure Association (5G IA), "European Vision for the 6G Network Ecosystem," Jun. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5007671

[21] P. Lincoln, A. Blate, M. Singh, T. Whitted, A. State, A. Lastra, and H. Fuchs, "From motion to photons in 80 microseconds: Towards minimal latency for virtual and augmented reality," *IEEE Transactions on Visualization and Computer Graphics*, vol. 22, no. 4, pp. 1367–1376, 2016.

[22] E. Uhlemann, "Time for autonomous vehicles to connect [connected vehicles]," *IEEE Vehicular Technology Magazine*, vol. 13, pp. 10–13, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:52150199

[23] Intel, "Tofino Programmable Ethernet Switch ASIC," 2016. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[24] "NVIDIA BlueField Networking Platform," https://shorturl.at/fmDKO, NVIDIA Networking, Mountain View, CA, United States.

[25] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, jul 2014.

[26] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with p4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804522002028

[27] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[28] D. R. K. Ports and J. Nelson, "When should the network be the computer?" ser. HotOS '19. NY, USA: ACM, 2019, pp. 209–215.

[29] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, ser. HotNets'17, 2017.

[30] R. Parizotto, B. L. Coelho, D. C. Nunes, I. Haque, and A. Schaeffer-Filho, "Offloading machine learning to programmable data planes: A systematic survey," *ACM Comput. Surv.*, vol. 56, no. 1, 2023.

[31] C. Zheng, X. Hong, D. Ding, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "In-network machine learning using programmable network devices: A survey," *IEEE Commun. Surv. Tutor.*, 2023.

[32] D. Barradas, N. Santos, L. Rodrigues, S. Signorello, F. M. V. Ramos, and A. Madeira, "Flowlens: Enabling efficient flow classification for ML-based network security applications," in *NDSS*, 2021. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/ndss2021_7C-2_24067_paper.pdf

[33] J. a. Romeiras Amado, F. C. Pereira, S. Signorello, M. Correia, and F. Ramos, "Poster: In-network ML feature computation for malicious traffic detection," in *Proceedings of the ACM SIGCOMM 2023 Conference*. New York, NY, USA: Association for Computing Machinery, 2023, pp. 1105–1107. [Online]. Available: https://doi.org/10.1145/3603269.3610866

[34] J. R. Amado, F. Pereira, D. Pissarra, S. Signorello, M. Correia, and F. M. V. Ramos, "Peregrine: Ml-based malicious traffic detection for terabit networks," 2024.

[35] M. Seufert, K. Dietz, N. Wehner, S. Geißler, J. Schüler, M. Wolz, A. Hotho, P. Casas, T. Hoßfeld, and A. Feldmann, "Marina: Realizing ml-driven real-time network traffic monitoring at terabit scale," *IEEE Transactions on Network and Service Management*, pp. 1–1, 2024.

[36] G. Siracusano and R. Bifulco, "In-network neural networks," *CoRR*, 2018.

[37] M. Courbariaux, Y. Bengio, and J.-P. David, "Binaryconnect: Training deep neural networks with binary weights during propagations," in *Advances in Neural Information Processing Systems*, vol. 28. Curran Associates, Inc., 2015.

[38] Z. Xiong and N. Zilberman, "Do switches dream of machine learning? toward in-network classification," in *HotNets 2019*. NY, USA: ACM, 2019, p. 25â33.

[39] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "IIsy: Practical in-network classification," *arXiv*, 2022.

[40] C. Zheng, M. Zang, X. Hong, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Automating in-network machine learning," *arXiv*, 2022.

[41] C. Busse-Grawitz, R. Meier, A. Dietmüller, T. Bühler, and L. Vanbever, "pForest: In-network inference with random forests," *CoRR*, 2019.

[42] J. Lee and K. P. Singh, "Switchtree: in-network computing and traffic analyses with random forests," *Neural Computing and Applications*, pp. 1–12, 2020.

[43] C. Zheng and N. Zilberman, "Planter: Seeding trees within switches," in *SIGCOMM '21*. NY, USA: ACM, 2021, pp. 12–14.

[44] X. Zhang, L. Cui, F. P. Tso, and W. Jia, "pHeavy: Predicting heavy flows in the programmable data plane," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, no. 4, pp. 4353–4364, 2021.

[45] P4.org, "P4-16 Language Specification." [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.1.html#sec-varbit-string

[46] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?" in *NeurIPS 2022*, ser. Advances in Neural Information Processing, New Orleans, United States, Nov. 2022. [Online]. Available: https://hal.science/hal-03723551

[47] Netronome, "Netronome Agilio SmartNICs," 2016. [Online]. Available: https://www.netronome.com/products/smartnic/overview/

[48] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K.-J. Grinnemo, P. Hurtig, N. Khademi, M. Tüxen, M. Welzl, D. Damjanovic, and S. Mangiante, "De-ossifying the internet transport layer: A survey and future perspectives," *IEEE Communications Surveys Tutorials*, vol. 19, no. 1, pp. 619–639, 2017.

[49] N. Feamster, J. Rexford, and E. Zegura, "The road to SDN: An intellectual history of programmable networks," *Queue*, vol. 11, no. 12, pp. 20–40, 2013.

[50] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *SIGCOMM Comput. Commun. Rev.*, vol. 26, no. 2, pp. 5–17, 1996.

[51] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," vol. 38, no. 2, pp. 69–74, 2008.

[52] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "Nox: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, jul 2008. [Online]. Available: https://doi.org/10.1145/1384609.1384625

[53] K. Nisar, E. R. Jimson, M. H. A. Hijazi, I. Welch, R. Hassan, A. H. M. Aman, A. H. Sodhro, S. Pirbhulal, and S. Khan, "A survey on the architecture, application, and security of software defined networking: Challenges and open issues," *Internet of Things*, vol. 12, p. 100289, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2542660520301219

[54] J. Xu, K. Papangelis, J. Dunham, J. Goncalves, N. J. LaLone, A. Chamberlain, I. Lykourentzou, F. L. Vinella, and D. I. Schwartz, "Metaverse: The vision for the future," in *CHI EA '22*. NY, USA: ACM, 2022.

[55] M. E. Kanakis, R. Khalili, and L. Wang, "Machine learning for computer systems and networking: A survey," *ACM Comput. Surv.*, Feb 2022.

[56] Y. Zhao, Y. Li, X. Zhang, G. Geng, W. Zhang, and Y. Sun, "A survey of networking applications applying the software defined networking concept based on machine learning," *IEEE Access*, vol. 7, 2019.

[57] J. Xie, F. R. Yu, T. Huang, R. Xie, J. Liu, C. Wang, and Y. Liu, "A survey of machine learning techniques applied to software defined networking (SDN): Research issues and challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 393–430, 2019.

[58] A. A. Gebremariam, M. Usman, and M. Qaraqe, "Applications of artificial intelligence and machine learning in the area of SDN and NFV: A survey," *SSD 2019*, pp. 545–549, 2019.

[59] A. Sapio, M. Canini, C.-Y. Ho, J. Nelson, P. Kalnis, C. Kim, A. Krishnamurthy, M. Moshref, D. Ports, and P. Richtarik, "Scaling distributed machine learning with In-Network aggregation," in *18th NSDI*. USENIX, Apr. 2021, pp. 785–808.

[60] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the AI accelerator?" *NetCompute '18*, 2018.

[61] A. Mestres, A. Rodriguez-Natal, J. Carner, P. Barlet-Ros, E. Alarcón, M. Solé, V. Muntés-Mulero, D. Meyer, S. Barkai, M. J. Hibbett, G. Estrada, K. Ma'ruf,

F. Coras, V. Ermagan, H. Latapie, C. Cassar, J. Evans, F. Maino, J. Walrand, and A. Cabellos, "Knowledge-defined networking," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 3, pp. 2–10, sep 2017.

[62] T. A. Benson, "In-network compute: Considered armed and dangerous," ser. HotOS '19. NY, USA: ACM, 2019, pp. 216–224.

[63] L. Bracciale, T. Swamy, M. Shahbaz, P. Loreti, S. Salsano, and H. Elbakoury, "The case for native multi-node in-network machine learning," ser. NativeNi '22. NY, USA: ACM, 2022, pp. 8–13. [Online]. Available: https://doi.org/10.1145/3565009.3569524

[64] G. Xie, Q. Li, G. Duan, J. Lin, Y. Dong, Y. Jiang, D. Zhao, and Y. Yang, "Empowering in-network classification in programmable switches by binary decision tree and knowledge distillation," *IEEE/ACM Trans. Netw.*, 2023.

[65] X. Zhang, L. Cui, F. P. Tso, W. Li, and W. Jia, "IN3: A framework for in-network computation of neural networks in the programmable data plane," *IEEE Communications Magazine*, vol. 62, no. 4, pp. 96–102, 2024.

[66] E. Paolini, L. D. Marinis, D. Scano, and F. Paolucci, "In-line any-depth deep neural networks using p4 switches," *IEEE Open Journal of the Communications Society*, pp. 1–1, 2024.

[67] Z. Zhao, Z. Li, Z. Song, F. Zhang, and B. Chen, "Rids: Towards advanced ids via rnn model and programmable switches co-designed approaches," in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, 2024.

[68] J. Yan, H. Xu, Z. Liu, Q. Li, K. Xu, M. Xu, and J. Wu, "Brain-on-Switch: Towards advanced intelligent network data plane via NN-Driven traffic analysis at Line-Speed," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 419–440. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/yan

[69] G. Siracusano, S. Galea, D. Sanvito, M. Malekzadeh, H. Haddadi, G. Antichi, and R. Bifulco, "Re-architecting traffic analysis with neural network interface cards," in *NSDI*. Renton, WA: USENIX, Apr. 2022.

[70] N. Corporation, "ConnectX SmartNICs - 10/25/40/50/100/200 and 400G Ethernet Network Adapters," 2022. [Online]. Available: https://www.nvidia.com/en-gb/networking/ethernet-adapters/

[71] T. Swamy, A. Rucker, M. Shahbaz, I. Gaur, and K. Olukotun, "Taurus: A data plane architecture for per-packet ML," *ASPLOS*, 2022.

[72] A. Monterubbiano, R. Azorin, G. Castellano, M. Gallo, S. Pontarelli, and D. Rossi, "Memory-efficient random forests in fpga smartnics," in *Companion of the 19th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 55â56. [Online]. Available: https://doi.org/10.1145/3624354.3630089

[73] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in python," *J. Mach. Learn. Res.*, vol. 12, p. 2825â2830, nov 2011.

[74] T. A. Tang, L. Mhamdi, D. McLernon, S. A. R. Zaidi, and M. Ghogho, "Deep learning approach for network intrusion detection in software defined networking," in *2016 WINCOM*, 2016.

[75] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org.

[76] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to + 1 or -1," *arXiv preprint arXiv:1602.02830*, 2016.

[77] L. Geiger and P. Team, "Larq: An open-source library for training binarized neural networks," *Journal of Open Source Software*, vol. 5, no. 45, Jan. 2020.

[78] A. Sivanathan, H. H. Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT devices in smart environments using network traffic characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, 2019.

[79] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli, "Detection of encrypted tunnels across network boundaries," *2008 IEEE ICC*, pp. 1738–1744, 2008.

[80] A. Este, F. Gringoli, and L. Salgarelli, "On-line SVM traffic classification," in *2011 7th IWCMC*, 2011.

[81] I. Sharafaldin, A. Habibi Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization," *ICISSP 2018*, pp. 108–116, 2018.

[82] N. Moustafa and J. Slay, "UNSW-NB15: a comprehensive data set for network intrusion detection systems (UNSW-NB15 network data set)," in *MilCIS 2015)*, 2015.

[83] ——, "The evaluation of network anomaly detection systems: Statistical analysis of the UNSW-NB15 data set and the comparison with the KDD99 data set," *Inf. Sec. J.: A Global Perspective*, vol. 25, no. 1â3, 2016.

[84] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the KDD CUP 99 data set," in *2009 IEEE CISDA*, 2009.

[85] B. Hullar, S. Laki, and A. Gyorgy, "Early identification of peer-to-peer traffic," in *2011 IEEE International Conference on Communications (ICC)*, 2011.

[86] F. Pacheco, E. Exposito, M. Gineste, C. Baudoin, and J. Aguilar, "Towards the deployment of machine learning solutions in network traffic classification: A systematic survey," *IEEE Communications Surveys Tutorials*, vol. 21, no. 2, 2019.

[87] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett, Eds., vol. 29. Curran Associates, Inc., 2016.

[88] X. Lin, C. Zhao, and W. Pan, "Towards accurate binary convolutional neural network," *31st ICANIPS*, 2017.

[89] C. Zheng, Z. Xiong, T. T. Bui, S. Kaupmees, R. Bensoussane, A. Bernabeu, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Iisy: Hybrid in-network classification using programmable switches," *IEEE/ACM Transactions on Networking*, pp. 1–16, 2024.

[90] C. Zheng, M. Zang, X. Hong, L. Perreault, R. Bensoussane, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "Planter: Rapid prototyping of in-network machine learning inference," *ACM SIGCOMM Computer Communication Review*, 2024.

[91] B. M. Xavier, R. S. Guimarães, G. Comarela, and M. Martinello, "Programmable switches for in-networking classification," in *IEEE INFOCOM*, 2021.

[92] B. M. Xavier, R. Silva Guimarães, G. Comarela, and M. Martinello, "MAP4: A pragmatic framework for in-network machine learning traffic classification," *IEEE Trans. Netw. Service Manag*, vol. 19, no. 4, 2022.

[93] G. Xie, Q. Li, Y. Dong, G. Duan, Y. Jiang, and J. Duan, "Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation," in *IEEE INFOCOM 2022*, 2022, pp. 1938–1947.

[94] B. Coelho and A. Schaeffer-Filho, "BACKORDERS: Using random forests to detect DDoS attacks in programmable data planes," in *EuroP4 '22*. NY, USA: ACM, 2022, pp. 1–7.

[95] H. Siddique, M. Neves, C. Kuzniar, and I. Haque, "Towards network-accelerated ML-based distributed computer vision systems," in *IEEE ICPADS*, 2021.

[96] G. Xie, Q. Li, C. Cui, P. Zhu, D. Zhao, W. Shi, Z. Qi, Y. Jiang, and X. Xiao, "Soter: Deep learning enhanced in-network attack detection based on programmable switches," in *2022 41st International Symposium on Reliable Distributed Systems*, 2022.

[97] K. Friday, E. Kfoury, E. Bou-Harb, and J. Crichigno, "INC: In-network classification of botnet propagation at line rate," in *Computer Security – ESORICS*, 2022.

[98] K. Friday, E. Bou-Harb, and J. Crichigno, "A learning methodology for line-rate ransomware mitigation with P4 switches," in *NSS*, 2022.

[99] R. Li, Q. Li, Y. Zhang, D. Zhao, X. Xiao, and Y. Jiang, "Genos: General in-network unsupervised intrusion detection by rule extraction," in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, 2024.

[100] S. U. Jafri, S. Rao, V. Shrivastav, and M. Tawarmalani, "Leo: Online ML-based traffic classification at Multi-Terabit line rate," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*. Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1573–1591. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/jafri

[101] G. Zhou, Z. Liu, C. Fu, Q. Li, and K. Xu, "An efficient design of intelligent network data plane," in *USENIX Security 23*, 2023. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zhou-guangmeng

[102] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian, "Traffic classification on the fly," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 2, pp. 23–26, 2006.

[103] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 5–16, 2007.

[104] L. Bernaille and R. Teixeira, "Early recognition of encrypted applications," in *PAM*. Springer, 2007, pp. 165–175.

[105] M. Jaber, R. G. Cascella, and C. Barakat, "Can we trust the inter-packet time for traffic classification?" in *IEEE ICC*, 2011, pp. 1–5.

[106] G. Lu, H. Zhang, M. Qassrawi, and X. Yu, "Comparison and analysis of flow features at the packet level for traffic classification," in *ICCVE*, 2012, pp. 262–267.

[107] R. Panigrahy and S. Sharma, "Reducing TCAM power consumption and increasing throughput," in *Proceedings 10th Symposium on High Performance Interconnects*, 2002, pp. 107–112.

[108] H. Kim, X. Chen, J. Brassil, and J. Rexford, "Experience-driven research on programmable networks," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 1, pp. 10–17, 2021.

[109] A. Turner and F. Klassen, "Tcpreplay," 2013. [Online]. Available: https://tcpreplay.appneta.com/

[110] G. Combs, "Tshark," *Wireshark*, 1998. [Online]. Available: https://www.wireshark.org/docs/man-pages/tshark.html

[111] Open Networking Foundation, "P4Runtime Specification," https://opennetworking.org/wp-content/uploads/2020/10/P4Runtime-Specification-120.html, [Online; accessed 24-February-2024].

[112] Intel Corporation, "SDE 9.13.0 BF-Runtime gRPC Doxygen," https://www.intel.com/content/www/us/en/secure/content-details/778340/sde-9-13-0-bf-runtime-grpc-doxygen.html, [Online; accessed 24-February-2024].

[113] ——, "P4_16 Intel Tofino Native Architecture - Public Version," https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf, 2021.

[114] P4 Language Consortium, "Behavioral model (bmv2)," 2018. [Online]. Available: https://github.com/p4lang/behavioral-model

[115] Mininet Team, "Mininet: An instant virtual network on your laptop (or other pc)," 2017. [Online]. Available: http://mininet.org/

[116] Intel, "P4 Insight," https://www.intel.com/content/www/us/en/products/details/network-io/intelligent-fabric-processors/p4-insight.html.

[117] "Tcpdump." [Online]. Available: https://www.tcpdump.org/

[118] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *IMC*, 2015.

[119] S. Garcia, A. Parmisano, and M. J. Erquiaga, "IoT-23: A labeled dataset with malicious and benign IoT network traffic," Zenodo, 2020. [Online]. Available: http://doi.org/10.5281/zenodo.4743746

[120] A. Alsaedi, N. Moustafa, Z. Tari, A. Mahmood, and A. Anwar, "TON_IoT telemetry dataset: A new generation dataset of IoT and IIoT for data-driven intrusion detection systems," *IEEE Access*, vol. 8, 2020.

[121] Z. Erdenebaatar, B. Nandy, N. Seddigh, R. Alshammari, M. Elsayed, and N. Zincir-Heywood, "Instant messaging application encrypted traffic generation system," in *IEEE/IFIP NOMS*, 2023.

[122] Z. Erdenebaatar, "Encrypted Mobile Instant Messaging Traffic Dataset." [Online]. Available: https://shorturl.at/gpw02

[123] V. Tong, H. A. Tran, S. Souihi, and A. Mellouk, "A novel QUIC traffic classifier based on convolutional neural networks," in *IEEE GLOBECOM*, 2018.

[124] I. Akbari, M. A. Salahuddin, L. Ven, N. Limam, R. Boutaba, B. Mathieu, S. Moteau, and S. Tuffin, "Traffic classification in an increasingly encrypted web," *Communications of the ACM*, vol. 65, 2022.

[125] G. Draper-Gil, A. H. Lashkari, M. S. I. Mamun, and A. A. Ghorbani, "Characterization of encrypted and VPN traffic using time-related features," in *ICISSP*, 2016.

[126] P. Radoglou-Grammatikis, V. Kelli, T. Lagkas, V. Argyriou, and P. Sarigiannidis, "DNP3 intrusion detection dataset," 2022. [Online]. Available: https://dx.doi.org/10.21227/s7h0-b081

[127] V. Kelli, P. Radoglou-Grammatikis, T. Lagkas, E. K. Markakis, and P. Sarigiannidis, "Risk analysis of DNP3 attacks," in *2022 IEEE International Conference on Cyber Security and Resilience (CSR)*, 2022, pp. 351–356.

[128] V. Kelli, P. Radoglou-Grammatikis, A. Sesis, T. Lagkas, E. Fountoukidis, E. Kafetzakis, I. Giannoulakis, and P. Sarigiannidis, "Attacking and defending DNP3 ICS/SCADA systems," in *2022 18th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2022, pp. 183–190.

[129] S. East, J. Butts, M. Papa, and S. Shenoi, "A taxonomy of attacks on the DNP3 protocol," in *Critical Infrastructure Protection III*, C. Palmer and S. Shenoi, Eds., 2009, pp. 67–81.

[130] K. Proska, J. Wolfram, J. Wilson, D. Black, K. Lunden, D. K. Zafra, N. Brubaker, T. McLellan, and C. Sistrunk, "Sandworm disrupts power in Ukraine using a novel

attack against operational technology," Mandiant: https://www.mandiant.com/resources/blog/sandworm-disrupts-power-ukraine-operational-technology, 11 2023.

[131] INSIKT GROUP, "China-linked group redecho targets the indian power sector amid heightened border tensions," Recorded Future: https://www.recordedfuture.com/redecho-targeting-indian-power-sector, 2 2021.

[132] M. Z. Gunduz and R. Das, "Cyber-security on smart grid: Threats and potential solutions," *Computer Networks*, vol. 169, 2020.

[133] D. Mashima, Y. Chen, M. M. Roomi, S. Lakshminarayana, and D. Chen, "Cybersecurity for modern smart grid against emerging threats," *Foundations and Trends in Privacy and Security*, vol. 5, no. 4, pp. 189–285, 2023.

[134] D. Pliatsios, P. Sarigiannidis, T. Lagkas, and A. G. Sarigiannidis, "A survey on SCADA systems: Secure protocols, incidents, threats and tactics," *IEEE Communications Surveys Tutorials*, vol. 22, no. 3, 2020.

[135] P. Vargas and I. Tien, "Impacts of 5G on cyber-physical risks for interdependent connected smart critical infrastructure systems," *International Journal of Critical Infrastructure Protection*, vol. 42, 2023.

[136] J. Jow, X. Yang, and W. Han, "A survey of intrusion detection systems in smart grid," *International Journal of Sensor Networks*, vol. 23, 2017.

[137] T. Berghout, M. Benbouzid, and S. M. Muyeen, "Machine learning for cybersecurity in smart grids: A comprehensive review-based study on methods, solutions, and prospects," *International Journal of Critical Infrastructure Protection*, vol. 38, 2022.

[138] A. Iqbal and Pooja, "Intrusion detection in smart grid using machine learning approach," *Journal of Computational and Theoretical Nanoscience*, vol. 16, 2019.

[139] W. Xiong, Y. Lei, Y. Zhou, Y. Zhang, S. Wei, and Z. Liu, "A smart grid traffic anomaly detector based on deep learning," in *2022 International Conference on Frontiers of Communications, Information System and Data Science (CISDS)*, 2022.

[140] J. Cao, D. Wang, Z. Qu, M. Cui, P. Xu, K. Xue, and K. Hu, "A novel false data injection attack detection model of the cyber-physical power system," *IEEE Access*, vol. 8, 2020. [Online]. Available: https://shorturl.at/lmVW3

[141] I. Siniosoglou, P. Radoglou-Grammatikis, G. Efstathopoulos, P. Fouliras, and P. Sarigiannidis, "A unified deep learning anomaly detection and classification approach for smart grid environments," *IEEE Trans. Netw. Serv. Manag.*, vol. 18, 2021. [Online]. Available: https://shorturl.at/lpHSW

[142] F. Holik and P. Dolezel, "Industrial network protection by SDN-based IPS with AI," *Communications in Computer and Information Science*, vol. 1178 CCIS, 2020.

[143] Z. A. E. Houda, B. Brik, and L. Khoukhi, "Ensemble learning for intrusion detection in sdn-based zero touch smart grid systems," in *IEEE LCN*, 2022.

[144] P. Radoglou-Grammatikis, P. Sarigiannidis, G. Efstathopoulos, P.-A. Karypidis, and A. Sarigiannidis, "DIDEROT: an intrusion detection and prevention system for DNP3-based SCADA systems," in *ARES*, 2020.

[145] G. K. Ndonda and R. Sadre, "A two-level intrusion detection system for industrial control system networks using P4," in *5th International Symposium for ICS SCADA Cyber Security Research 2018 (ICS-CSR)*, 2018.

[146] M. H. Rehmani, F. Akhtar, A. Davy, and B. Jennings, "Achieving resilience in SDN-based smart grid: A multi-armed bandit approach," in *IEEE NetSoft*, 2018.

[147] S. M. Taghavinejad, M. Taghavinejad, L. Shahmiri, M. Zavvar, and M. H. Zavvar, "Intrusion detection in IoT-based smart grid using hybrid decision tree," in *ICWR*, 2020.

[148] H. He, K. Khoshelham, and C. Fraser, "A two-step classification approach to distinguishing similar objects in mobile LIDAR point clouds," vol. IV-2/W4, 09 2017, pp. 67–74.

[149] R. Babbar, I. Partalas, E. Gaussier, and M.-R. Amini, "On flat versus hierarchical classification in large-scale taxonomies," in *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'13, Red Hook, NY, USA, 2013, pp. 1824–1832.

[150] C. N. Silla and A. A. Freitas, "A survey of hierarchical classification across different application domains," *Data Mining and Knowledge Discovery*, vol. 22, 2011.

[151] M. Hayes, B. Ng, A. Pekar, and W. K. Seah, "Scalable architecture for SDN traffic classification," *IEEE Systems Journal*, vol. 12, 2018.

[152] L. Peng, B. Yang, Y. Chen, and Z. Chen, "Effectiveness of statistical features for early stage internet traffic identification," *Int. J. Parallel Program.*, vol. 44, 2016.

[153] X. Jin, L. E. Li, L. Vanbever, and J. Rexford, "Softcell: Scalable and flexible cellular core network architecture," in *ACM CoNEXT*, 2013.

[154] S. C. Madanapalli, A. Mathai, H. H. Gharakheili, and V. Sivaraman, "Reclive: Real-time classification and QoE inference of live video streaming services," in *IEEE/ACM IWQOS*, 2021.

[155] Y. Zeng and T. M. Chen, "Classification of traffic flows into QoS classes by unsupervised learning and KNN clustering," *KSII Transactions on Internet and Information Systems*, vol. 3, 2009. [Online]. Available: https://doi.org/10.3837/TIIS.2009.02.001

[156] F. Mola and R. Siciliano, "A fast splitting procedure for classification trees," *Statistics and Computing*, vol. 7, pp. 209–216, 1997. [Online]. Available: https://doi.org/10.1023/A:1018590219790

[157] Palo Alto Networks Unit 42, "2020 unit 42 IoT threat report," 2020. [Online]. Available: https://unit42.paloaltonetworks.com/iot-threat-report-2020/

[158] A.T-J. Akem et al., "Henna," https://github.com/nds-group/Henna.

[159] G. Xie et al., "Mousika," https://github.com/xgr19/Mousika.

[160] X. Gao et al., "Soter," https://github.com/xgr19/Soter/tree/main.

[161] G. Zhou et al., "Netbeacon," https://github.com/IDP-code/NetBeacon.

[162] Telecommunication Networks Group at the University of Brescia, "UNIBS internet traces," http://netweb.ing.unibs.it/~ntw/tools/traces/, 2009, accessed: 2021-12-11.