TÉCNICO LISBOA

In-network ML-based Anomaly Detection

David de Almeida Pissarra

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Fernando Manuel Valente Ramos
Prof. Muhammad Shahbaz

Examination Committee

Chairperson: Prof. Alberto Manuel Rodrigues da Silva
Supervisor: Prof. Fernando Manuel Valente Ramos
Member of the Committee: Prof. Paolo Romano

November 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to express my gratitude to the individuals who have contributed to the completion of this thesis. First and foremost, I extend my appreciation to my thesis advisors, Prof. Fernando Ramos and Prof. Muhammad Shahbaz. Their expertise and invaluable feedback were crucial in shaping this work. A special thanks to Francisco, João and Salvatore for the insightful discussions and help throughout this journey. I would like to thank my family and my friends for always supporting me. Finally, I want to thank Lara for always being by my side.

Without **all** of you, this thesis would not be possible.

This work was supported by the European Union (ACES project, 101093126).

Abstract

The progress made in anomaly detection mainly owes to the breakthrough of recent Machine Learning (ML) algorithms, enabling accurate Network Intrusion Detection Systems (NIDS). Though, even cutting-edge network devices, such as programmable switches, lack the necessary computation to accelerate such algorithms in the network data plane. Existing alternatives offload these models to server-based middleboxes but generally lose on throughput, and cannot achieve line-rate speeds. On the other hand, to be able to achieve line-rate while detecting some of the anomalies, network operators still implement simple heuristics in the fast path that fail to understand unseen behavior, such as *zero-day* attacks, or either deploy simple ML models that cause systematic false positives. As a consequence, running complex decision-making at high throughputs is a big challenge. State-of-the-art already explores programmable switches to perform feature computation entirely in the data plane. However, since a recent extension to the now common PISA architecture of current programmable switches allows complex arithmetic operations, we move forward and run also the ML inference in the switch, enabling detection at packet time scales. The main challenge is that running NIDS in the switch is still limited to small ML models. To address this problem we explore different DNN models, and some techniques to further optimize their execution.

Keywords

In-Network Anomaly Detection; Network Intrusion Detection Systems; Per-Packet ML; Programmable Switches; Data Plane ML.

Resumo

O progresso alcançado na deteção de anomalias deve-se principalmente aos recentes avanços de algoritmos de aprendizagem, possibilitando soluções em sistemas de deteção de intrusão de rede mais precisas. Embora mesmo dispositivos de rede de última geração, como *switches* programáveis, não possuem a capacidade computacional necessária para executar tais algoritmos no plano de dados da rede. Alternativas existentes movem a computação destes modelos para *middleboxes*, mas geralmente perdem rendimento e não conseguem atingir velocidades à máxima taxa de transmissão. Por outro lado, de modo a atingir a taxa de transmissão e manter a deteção de anomalias, operadores de rede ainda implementam nas redes de computadores heurísticas simples que falham em perceber comportamentos inesperados, tal como ataques dia zero, ou então recorrem a simples modelos de aprendizagem que causam frequentes falsos positivos. Deste modo, executar tomada de decisão com base nestes modelos a alto rendimento é bastante desafiante. O atual estado da arte explora *switches* programáveis para calcular estatísticas da rede somente no plano de dados. No entanto, uma nova extensão à comum arquitetura de *switches* programáveis, PISA, permite a execução de aritmética complexa. Aproveitamos esta oportunidade para também executar a inferência de algoritmos de aprendizagem no *switch*, e realizar a sua execução à taxa de transmissão. O principal desafio é que implementar sistemas de deteção de intrusão diretamente em *switches* ainda é restrito a pequenos modelos de aprendizagem. De modo a encarar este problema, investigamos diferentes modelos baseados em redes neuronais, e adicionalmente alguns métodos de otimização desta gama de modelos.

Palavras Chave

Deteção de Anomalias em Redes; Sistemas de Deteção de Intrusão de Rede; Aprendizagem Pacote-a-Pacote; Switches Programáveis; Aprendizagem no Plano de Dados.

Contents

1 Introduction **1**

1.1 Problem & Motivation . . . 1

1.2 Opportunity . . . 3

1.3 Towards Data Plane ML . . . 4

1.4 Contributions . . . 4

1.5 Thesis Outline . . . 5

2 Background **7**

2.1 Programmable Networks . . . 7

2.1.1 Software Defined Networking . . . 7

2.1.2 Programmable Data Plane . . . 9

2.2 Anomaly Detection: An Overview . . . 10

2.3 Network Intrusion Detection Systems . . . 10

2.3.1 Misuse-based Intrusion Detection . . . 11

2.3.2 Anomaly-based Intrusion Detection . . . 11

2.4 Deep Neural Networks . . . 12

2.4.1 Autoencoders . . . 13

2.4.2 Variational Autoencoders . . . 14

2.4.3 Recurrent Neural Networks . . . 15

2.4.4 Gated Mechanisms . . . 15

2.4.5 Deep Gaussian Models . . . 17

2.5 Techniques to enable ML on constrained devices . . . 18

2.5.1 Feature Importance . . . 18

2.5.2 Quantization . . . 19

2.6 Summary . . . 22

3 Related Work **23**

3.1 ML in Networking . . . 23

3.2 The Challenge of ML-based Anomaly Detection . . . 24

3.3 Deploying ML Models in the Network Data Plane . . . 25
3.4 Per-Packet ML: Limitations & Opportunities . . . 25
3.4.1 Limitations . . . 25
3.4.2 Opportunity: A New Per-Packet ML Architecture . . . 27
3.5 ML Feature Computation in the Network Data Plane . . . 28
3.6 Summary . . . 29

4 Proposed Approach **31**
4.1 Representation Learning . . . 31
4.1.1 Model #1: Autoencoder . . . 31
4.1.2 Model #2: Gated Recurrent Autoencoder . . . 33
4.1.3 Model #3: Deep Gaussian Estimator . . . 34
4.2 Feature Importance . . . 36
4.2.1 Calculated Features . . . 36
4.2.2 Model Dimensionality . . . 36
4.3 Implementation . . . 37
4.3.1 Reconstruction-based Anomaly Detection . . . 38
4.3.2 Latent Analysis-based Anomaly Detection . . . 39
4.4 ML-based NIDS Workflow . . . 40
4.4.1 Offline Configuration Stage . . . 41
4.4.2 Online Runtime Stage . . . 42
4.5 Summary . . . 43

5 Experimental Results **45**
5.1 Implementation & Hardware . . . 45
5.2 Results & Analysis . . . 47
5.3 Feature Importance . . . 50
5.4 Model Dimensionality . . . 51
5.5 Quantization . . . 52
5.6 Discussion . . . 53

6 Conclusion **55**
6.1 Final Remarks . . . 55
6.2 Limitations & Future Work . . . 56

Bibliography **59**

A Overall Feature Importance **65**

List of Figures

2.1 Simplified view of an Software Defined Networking (SDN) architecture 8
2.2 Protocol Independent Switch Architecture (PISA) [Bosshart et al., 2014] 9
2.3 Traditional Autoencoder architecture 13
2.4 A traditional Recurrent Neural Network (RNN) 15
2.5 Latent vector mapped into a Gaussian Mixture Model (GMM) 17
2.6 Assymmetric range-based integer quantization 20
2.7 Quantize and Dequantize operations in Quantization Aware Training (QAT) 22

3.1 Decision Trees (DT)-based ensemble model directly mapped to the Protocol Independent Switch Architecture (PISA). Each level of each decision tree is transformed into match/action rules for further inference, within its respective pipeline stage. 26
3.2 Taurus architecture [Swamy et al., 2022a] 27
3.3 A three-stage CU, composed of Functional Units (FUs) and Pipeline Registers (PRs) [Swamy et al., 2022a]. 28

4.1 Our AE model vs. Kitsune [Mirsky et al., 2018] 32
4.2 Proposed Gated Recurrent Autoencoder workflow 34
4.3 Proposed Deep Gaussian architecture 35
4.4 ML-based NIDS proposed workflow 40

5.1 AUC-ROC score and Recall at FPR=0.01 comparison with Kitsune and the Taurus AD . . 48
5.2 Latent-space visualization for the *Botnet* attack 49
5.3 Latent-space visualization for the *SSDP Flood* attack with PCA 49
5.4 Performance degradation on *Port Scan* attack (from the best to the worst features) 50
5.5 Performance improvement on *DDoS HOIC* attack (from the worst to the best features) . . 50

A.1 Overall Feature Importance with XGBoost (XGB) [Chen and Guestrin, 2016] 66

List of Tables

3.1 ML-powered NIDS solutions comparison 29

4.1 Computed features [Mirsky et al., 2018] 37
4.2 Implemented models summary 37

5.1 Hardware consumption (fixed-point precision `fix8` and `fix32`) 47
5.2 Average relative improvement across the different attacks when selecting the *N* best features 51
5.3 Average relative degradation across the different attacks for different latent space dimensions dim($\mathbf{z}$) 52
5.4 Relative degradation across the different attacks with quantization, compared to full precision (32-bit) 53

Listings

4.1 Autoencoder (AE) implementation in Spatial [Koeplinger et al., 2018] 38
4.2 Deep Gaussian Estimator implementation in *PyTorch* 39

Acronyms

AE	Autoencoder
AD	Anomaly Detector
AI	Artificial Intelligence
API	Application Programming Interface
AUC	Area Under the Curve
BDT	Binary Decision Trees
BNN	Binary Neural Networks
CGRA	Coarse-Grained Reconfigurable Array
CU	Compute Unit
DGMM	Deep Gaussian Mixture Model
DL	Deep Learning
DP	Data Plane
DSL	Domain Specific Language
DT	Decision Trees
DNN	Deep Neural Network
FC	Feature Computation
FPGA	Field Programmable Gate Array
FU	Functional Unit
GMM	Gaussian Mixture Model
GRU	Gated Recurrent Unit
KL	Kullback-Leibler
LLMs	Large Language Models

LSTM	Long Short-Term Memory
MAT	Match-Action Tables
ML	Machine Learning
NIDS	Network Intrusion Detection Systems
PCA	Principal Component Analysis
PISA	Protocol Independent Switch Architecture
PR	Pipeline Register
PTQ	Post-Training Quantization
QAT	Quantization Aware Training
RF	Random Forest
RNN	Recurrent Neural Network
RMSE	Root Mean Squared Error
SDN	Software Defined Networking
SIMD	Single Instruction Multiple Data
VAE	Variational Autoencoder
VLIW	Very Long Instruction Word
XGB	XGBoost

Chapter 1

Introduction

As computer networks increasingly play a crucial role in our everyday activities, there is a growing need to safeguard users and network operators against potential cybersecurity threats. As these cyberattacks grow in complexity and frequency, traditional methods of detecting and mitigating intrusions often fall short. This is where Machine Learning (ML) has been stepping in, significantly improving the performance of Network Intrusion Detection Systems (NIDS). A challenge of these systems is that running ML models still represents a computational bottleneck. These models, especially Deep Learning (DL) algorithms, are computationally intensive, requiring significant processing power and memory bandwidth. Traditional network switching hardware cannot run ML computations, so NIDS are integrated with current networks as server-based middlebox appliances that run off the data path, requiring heavy packet sampling, thus, leading to slow inference times and reduced detection performance in practice. In this thesis, we leverage recent network switch architectures that are amenable to ML computations, to explore the potential of *In-network* ML-based Anomaly Detection.

1.1 Problem & Motivation

Due to the increasing use of Internet technologies, throughout the last three decades, networking systems are increasingly facing more security challenges. In order to ensure traffic security, networks implement detection systems to deal with increased traffic volumes and intrusion threats. The most common systems rely on simple metrics and *rules* extracted directly from the processed data [Zhao et al., 2020]. These signature-based NIDS work well when the threat patterns are well-defined and the data they are processing is representative of the types of network attacks they are designed to detect. However, these *rule-based* systems are not effective at detecting out-of-distribution cases, such as *zero-day* attacks, which involve patterns that are different, even if just slightly, from the patterns the detector was trained on. In contrast, data-driven approaches are more flexible, learning how to recognize a wider range of

patterns. These ML-based Anomaly Detection algorithms are therefore increasingly appealing for the networking systems field [Boutaba et al., 2018].

Anomaly detection has been a relevant topic in Artificial Intelligence (AI), with a broad research scope. Various types of anomaly detection systems have been permeating many important domains [Xu et al., 2018, Mahadevan et al., 2010], one of these being networking systems. Network infrastructures often include these middleboxes to automatically detect unusual traffic patterns, network behavior that might indicate a network issue or even a security threat. By detecting these anomalies in a timely manner, network administrators can take steps to prevent damage or disruptions within networks. However, since the Machine Learning algorithms that empower these systems can be computationally complex, it is a challenge to process network data in real-time [Fu et al., 2021]. This becomes a bottleneck in line-speed networks, and so, to guarantee the required throughputs, a trade-off between faster packet forwarding and complex decision-making must be taken into account. In this work, we take this challenge head-on, and look into Anomaly Detection applied to the field of NIDS, with focus on detecting malicious traffic on networks.

Despite the advantages of ML-based anomaly detectors, *rule-based* systems are still elected for large-scale deployments over ML-based detectors. In general, running *rule-based* detectors at *tbps* speeds remains a challenge [Zhao et al., 2020], but they still outperform ML-based detectors with regard to accuracy. Actually, *rule-based* detectors are very strong in detecting *known* attacks, avoiding false positives. In contrast, ML-based systems can acquire a general understanding of the network behavior, enabling the detection also of *unknown* attacks. However, these are not easily mapped to traditional switching hardware (e.g., programmable switches), due to their complexity. The recent efforts made by the research community on this topic proposed to directly map simple ML models, such as decision trees, on match/action pipelines [Coralie et al., 2019]. Others went further and encoded such models more efficiently [Zheng et al., 2022a, Zheng et al., 2022b, Xie et al., 2022, Zhou et al., 2023], consuming fewer pipeline stages. Still, lightweight ML algorithms are very sensitive when facing slight deviations from normal network patterns, generating many false alarms. Thus, more complex algorithms, such as Deep Learning models, can potentially improve the status quo.

The need to deploy Deep Learning models in NIDS is not new [Boutaba et al., 2018]. Particularly, Deep Learning can learn patterns and features that one may not find obvious, implementing multiple levels of abstraction that leverage accurate predictions. Two state-of-the-art systems, Kitsune [Mirsky et al., 2018] and Whisper [Fu et al., 2021], are server-based network anomaly detectors that implement deep autoencoders, and achieve good detection performance. Since these approaches offload their decision-making to the network control plane, they have limited throughput performance. More importantly, by running off the network path and at *gbps* packet processing speeds at best, they require heavy packet sampling (e.g., 1:1000 or more) to be practical. Alas, this dramatically degrades their detection

performance. On the other hand, running complex models directly in the network data plane is currently unfeasible.

1.2 Opportunity

Recently, there have been several research efforts to make the network data plane more flexible. Typically, increasing the complexity of the operations performed in the data plane implies reducing the throughput of the processed data. Simultaneously, strict security and modern service-level objectives motivate the need to execute more advanced routines on a per-packet basis. Indeed, recent breakthroughs on programmable switches, namely through the Protocol Independent Switch Architecture (PISA) architecture [Bosshart et al., 2014], give hope to evolve from a fixed function forwarding pipeline to a flexible and controllable data plane that can be used to implement complex workloads on packets at line rate speeds. Despite the advancements that brought the PISA switching pipeline, programmable switches are not Turing Complete. To guarantee terabit throughputs, they need to process packets deterministically in a few nanoseconds. They therefore cannot perform conventional operations in CPUs (e.g., multiplication, division), loops are either inexistent or very limited, and computations are restricted to simple arithmetic operations (e.g., addition, subtraction, bit shifts). This creates significant barriers in terms of the computations required by ML models. For this reason, the research community is investigating new hardware to serve the purpose of running complex decision-making.

Some recent approaches have started dodging the limitations of the data plane. As a matter of fact, Peregrine [Amado et al., 2023] moves the entire feature computation module to the network data plane, computing flow-level features with per-packet granularity. These features are then summarized into reports that are periodically sent to the control plane to be checked by the ML model, where detection inference takes place. Despite Peregrine still not performing ML inference in the fast path, the input to the model includes statistical information computed on every packet at line-rate without occurring sampling loss. To further aid performant complex decision-making in the network, Taurus [Swamy et al., 2022a] introduced a new switch architecture that includes primitives to execute ML-based algorithms. This is the missing piece to enable ML-based anomaly detection *entirely* in the network data plane. To achieve high-detection performance at line-rate, our goal is to run the computation of an anomaly detector — both feature computation [Amado et al., 2023] and ML inference — entirely in the data plane, exploring traditional Protocol Independent Switch Architecture (PISA) switches [Bosshart et al., 2014], and new switch architectures, such as Taurus [Swamy et al., 2022a], to assist complex ML inference.

1.3 Towards Data Plane ML

Deploying a detection system in a demanding and resource-constrained environment, such as the network data plane, is not trivial. For instance, NIDS are required to meet high performance to effectively process data at line-rate speeds. As a result, besides exploring recent switching designs that run alongside a PISA pipeline to enable both terabit packet processing and per-packet ML inference [Swamy et al., 2022a], in this thesis we also delve into other techniques that can provide a compressed and optimized execution of ML workloads, such as quantization [Wu et al., 2020]. In this way, we leverage other important related fields to promote ML in the network data plane.

The term *quantization* refers to the process of mapping continuous values to a set of discrete finite ones. For some, the drive behind quantization arises from findings in neuroscience indicating that the human brain encodes information in a discrete, quantized manner rather than continuous [Tee and Taylor, 2020]. In the context of ML, this process typically involves converting the pre-trained model representations to a lower and compact bit-width, that encodes the crucial weight signals, so that model computations are performed at a cheaper cost. Therefore, quantization can be seen as the approximation of continuous values, typically which use a floating point representation, to scales that can resort to a very limited number of bits, for instance the usual integer representation or the fixed point representation. Despite the fact that approximating such intermediate results can lead to precision degradation of the ML model, the task for quantization algorithms is to introduce little performance loss while speeding up model inference. For some fields such as generative networks, quantization has truly been revolutionary [Frantar et al., 2023], allowing to maintain the overall model knowledge acquired during the training phase, while being computationally inexpensive. In addition to quantization, a light representation learning model can provide compressed representations of data without requiring an extraordinary number of model parameters. This can be either achieved by performing a better selection of the extracted data features or by reducing the number of *hidden* features that are computed in intermediate model calculations. In this thesis, we will investigate the effect of quantization, and shrinking the model size on detection performance for the task of intrusion detection.

1.4 Contributions

The primary objective of this thesis is to demonstrate the feasibility of achieving performance levels comparable to Kitsune [Mirsky et al., 2018] while significantly reducing hardware resource consumption. We adapted existing lightweight models to evaluate their performance in the NIDS landscape. Subsequently, through empirical exploration, we introduced methods that compress these models while sustaining the anomaly detection performance. Additionally, we provided a comprehensive assessment of the resources required for executing these models within the Taurus architecture [Swamy et al., 2022a],

paving the way for running ML-based Anomaly Detection at line-rate.

Additionally, while the immediate application domain focuses on safeguarding network infrastructures, a broader aspiration of this research is to formulate a methodology that transcends the boundaries of network data and can be readily extended to various other sequential data types, encompassing domains such as financial transactions, IoT device monitoring, among other possible applications.

1.5 Thesis Outline

This document is structured as follows:

In Chapter 2, we initiate the document by presenting fundamental concepts that will serve as a foundation for the subsequent chapters. To begin, we lay the foundation by introducing programmable networks and their vital role. We then navigate through the realm of Network Intrusion Detection Systems (NIDS) before delving into the theory of Deep Neural Networks (DNNs). Finally, we wrap up with a comprehensive overview of methods to enable ML on devices with limited resources. Further, in Chapter 3 we explore the state-of-the-art in ML for networking, as well as a new architecture to enable performant ML inference in the data plane.

We proceed to Chapter 4, where we take a look at the models we explore, supported by their implementation. In addition, we present a ML workflow for NIDS. Moving on to Chapter 5, we present and discuss our experiments, comparing them with other NIDS. We also show the results of model compression techniques for this task.

We conclude our thesis in Chapter 6, by summarizing the main conclusions of this work, and discussing future directions within this topic.

Chapter 2

Background

In-network ML-based Anomaly Detection can today be considered an important and active research topic mainly owing to the success of Software Defined Networking (SDN) that allows network customization. Alongside, advancements in the field of ML, mainly through DNNs, have enabled the creation of interesting and innovative anomaly detectors in the field of Network Intrusion Detection Systems (NIDS). Since network devices are resource sensitive, the study of quantization also appears as a relevant research path in NIDS.

2.1 Programmable Networks

Traditionally, in conventional networking, the control plane and data plane were tightly integrated within network devices, making it challenging to adapt to changing network requirements without significant hardware modifications. Networks relied on fixed-function switches that were composed of switching hardware that simply forwarded packets based on a preconfigured unchanging set of rule types, which are populated by distributed protocols running in the network switch's control plane. However, to meet these evolving demands, the importance of introducing programmability over networks has caused a dramatic shift from traditional, static infrastructures to dynamic and agile network environments.

2.1.1 Software Defined Networking

The need to deploy custom routines over forwarded packets motivated the emergence of Software Defined Networking (SDN) [Kreutz et al., 2014]. SDN's innovation lies in abstracting the control plane, allowing network administrators to define network policies and behaviors through software, which can be dynamically applied to the underlying physical infrastructure. Thus, SDN introduces a logically centralized controller, which communicates with the switch in the data plane via a southbound communication

protocol such as OpenFlow [McKeown et al., 2008]. This decoupling of the control from the data plane (Figure 2.1) enables the network to become more programmable, adaptable, and responsive to the demands of modern applications and services, by enabling flexibility and customization over network hardware. In turn, this alleviates the cost of communicating with outside middleboxes or servers that were used to execute the required custom functions. Indeed, offloading computation results in higher overhead, lower throughput, and additional latency.

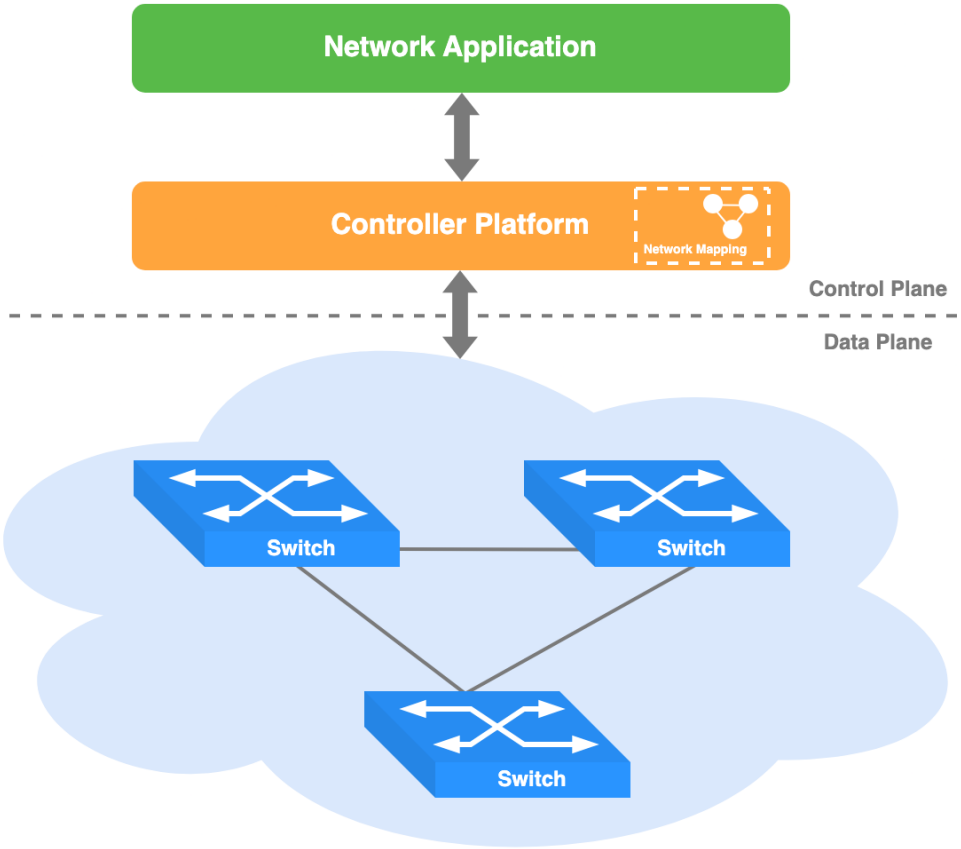

Figure 2.1: Simplified view of an SDN architecture

With this problem ahead, SDN came into play to make a clear decoupling between the control and data planes, by means of the introduction of an open interface between both, allowing remote control over the switch data plane. Briefly, the control plane has an overall vision of the network and defines the overall behavior of the network, whereas the data plane is the fast path that implements the behavior over individual packets (e.g., when to forward the packet). SDN architectures also promote open standards and APIs, fostering interoperability and encouraging innovation in the networking industry. As a result, SDN has not only revolutionized network management but has also paved the way for the development of novel networking applications, such as network slicing for 5G, intelligent traffic engineering, and intrusion

detection, the topic we explore next.

2.1.2 Programmable Data Plane

Later, after the introduction of SDN, network programmability was further extended with the development of programmable switches that enable programming also in the data plane. The emergence of domain-specific programming languages for network devices such as P4 [Bosshart et al., 2014], together with more robust compiler support, allowed the data plane to evolve from a fixed function forwarding pipeline to a more flexible data plane that can be reconfigured in the field to be able to modify the way the packets are forwarded. This emergence of programmable switches, owing to the success of Protocol Independent Switch Architecture (PISA) (e.g., P4 switches [Bosshart et al., 2014]), has created the opportunity to enable in-network intrusion detection, by increasing flexibility and control of the network data plane.

The PISA switching pipeline (Figure 2.2) is composed of a parser block for packet headers parsing, followed by multiple programmable match/action stages, that process packets given the implemented logic of a P4 program [Bosshart et al., 2014] (i.e., a language to program switch data planes). Nevertheless, to ensure line-rate speeds, these Match-Action Tabless (MATs) are very limited in terms of computational and memory resources. For instance, arithmetic logic in MATs is strictly limited in a way neither multiplication nor division is supported. Finally, on the last end of the pipeline, a deparser block serializes the outgoing packets. The existing limitations are the main reason why the research community has been working on relevant switch alternatives on top of the current PISA [Siracusano and Bifulco, 2018, Sanvito et al., 2018, Swamy et al., 2022a], one of which we describe later.

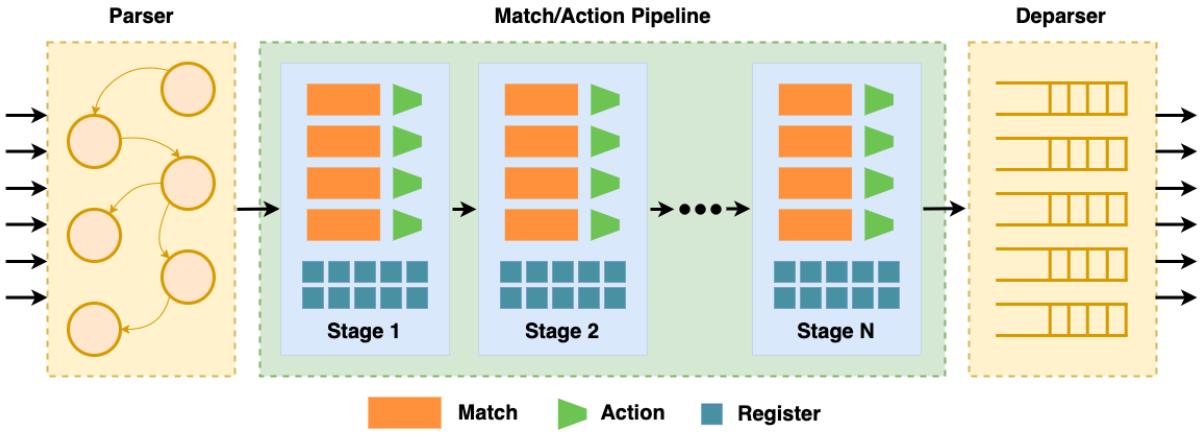

Figure 2.2: Protocol Independent Switch Architecture (PISA) [Bosshart et al., 2014]

2.2 Anomaly Detection: An Overview

Anomaly detection refers to the problem of finding patterns in data that differ from normal and expected behavior [Chandola et al., 2009]. Anomaly Detection can be often regarded as novelty detection or outlier detection, nevertheless, these designations are adjusted to slightly different tasks within the scope of Anomaly Detection. Whereas novelty detection focuses on disclosing new observation classes (i.e., non-existent in the training data), outlier detection is interested in detecting observations that are far from the normal patterns, typically in low density regions. Similarly to most NIDS, we formulate this work as an *outlier detection problem*, since we aim to distinguish anomalous deviations from what is considered to be normal traffic observations.

The breakthrough of anomaly detection in fields other than NIDS [Xu et al., 2018, Mahadevan et al., 2010], has enabled the use of thriving solutions in many other areas of research. More specifically, besides the successful applications of anomaly detection to monitor cybersecurity threats, it is also widely used to detect fraudulent financial transactions [Branco et al., 2020], product defects in the manufacturing industry [Quatrini et al., 2020], or even specific health conditions [Fernando et al., 2021]. These distinct kinds of applications end up intersecting together in the field of Machine Learning (ML), on which they can share significant insights. Thus, the heterogeneity of applications with ML-based anomaly detection algorithms is relevant and successfully promotes the progress of NIDS.

2.3 Network Intrusion Detection Systems

Identifying and preventing malicious activities within networks is currently one of the most critical requirements for every cutting-edge networking system. In order to achieve such condition, NIDS are employed to save organizations from imminent and recurrent security threats. Yet, it turns out that modern attacks are increasingly sophisticated, exploiting several limitations of existing NIDS. Thus, such attacks [Fu et al., 2021, Atre et al., 2022, Alcoz et al., 2022] have shown that the next NIDS should be robust enough to capture complex or even out-of-distribution threats.

Intrusion detection systems can be separated into two classes. In general, **Misuse-based Intrusion Detection** performs its detection using stored signatures of known attacks [Roesch et al., 1999, Paxson, 1999]. In contrast, **Anomaly-based Intrusion Detection** learns traffic patterns by observing the behavior of the network, having a prior notion of *normal behavior*, with the objective of detecting potential deviations from this normal pattern.

2.3.1 Misuse-based Intrusion Detection

Misuse-based or Signature-based NIDS detect attacks by matching network patterns against preconfigured signature profiles of known attacks. These signatures always rely on the knowledge of known attacks, being extracted directly from previous attacks or handcrafted by experts.

Two relevant works introducing signature-based NIDS were proposed by Roesch [Roesch et al., 1999] and Paxson [Paxson, 1999]. While *Snort* [Roesch et al., 1999] is centered around implementing a lightweight NIDS by importing rules operated on simple primitives, such as regular expressions and static strings, *Bro* [Paxson, 1999] further extends this approach by introducing a scripting language, which supported the expressiveness of the rulesets. As the size of the rulesets and the number of flows increases, neither approach is able to meet an acceptable performance on single servers. Thus, to operate networks at hundreds of *gbps* or even *tbps*, this requires a vast number of servers, in fact, motivating the creation of more robust architectures that could fulfill modern throughput requirements. Given the cost and complexity of this deployment, in practice, only a subset of traffic reaches the NIDS, by means of sampling, for instance.

The state-of-the-art signature-based NIDS [Zhao et al., 2020, Zhang et al., 2020, Liu et al., 2021] increase the throughput of the detection systems. Pigasus [Zhao et al., 2020] supports line rates on the order of 100 *gbps* while running hundreds of thousands of concurrent flows, capable of matching packets against tens of thousands of rules, by leveraging a Multi-String Pattern Matching layer implemented on a Field Programmable Gate Array (FPGA). Poseidon [Zhang et al., 2020] and Jaqen [Liu et al., 2021] go further, reaching line-speed (*tbps* rates) by employing programmable switches, but focusing only on DDoS, thus losing on generality. Despite the mentioned improvements, it is important to stress that misuse-based NIDS are dependent on the defined rulesets, and therefore are not able to detect unknown or zero-day attacks.

2.3.2 Anomaly-based Intrusion Detection

Anomaly detection has been emerging as an exciting research topic, due to its applicability to a wide diversity of fields, such as Time-Series data [Xu et al., 2018] or Computer Vision [Mahadevan et al., 2010]. By learning good representations of the respective data, these systems can generalize to unseen inputs, better recognizing possible divergences in data. In the context of NIDS, unlike misuse-based intrusion detectors, the anomaly-based class of NIDS focuses on acquiring general knowledge regarding normal network behavior, differentiating it from potential abnormal patterns. As a result, anomaly-based NIDS can generally detect new attack variations.

Typically, anomaly detection has been powered by the use of ML, and in particular the increasing use of DNNs [Boutaba et al., 2018] (we present some background on DNNs in the next section).

Anomaly-based NIDS can be broadly classified into two categories: flow-based and payload-based. While flow-based anomaly detection [Alcoz et al., 2022, Barradas et al., 2021, Mirsky et al., 2018] focuses on learning network patterns through flow features, payload-based anomaly detection [Boutaba et al., 2018] relies solely on computed features from the actual user data, in order to gauge its normality. However, due to its ability to map and aggregate potential malicious packets, current research is increasingly conducted towards flow feature-based anomaly detectors, that use enriched statistics to improve detection [Fu et al., 2021, Amado et al., 2023].

Nevertheless, in the field of NIDS, anomaly detection approaches are still often discarded for large-scale deployments due to some limitations. First, current anomaly detectors rely on complex algorithms, whose execution is demanding for resource-constrained network devices (e.g., network switches). Most approaches deploy the learned models on the network control plane [Mirsky et al., 2018, Hamza et al., 2019]. However, it is unfeasible to achieve line-rate/per-packet ML by offloading complex decision-making to the orders-of-magnitude slower control plane. Second, deployed anomaly-based models depend on continuous packet features, which are required to achieve robustness. Likewise, computing these features at line-rate remains a challenge, though feature computation has a crucial role in battling high false positive rates, a typical tendency by anomaly detectors. One may note that even low false-positive rates can result in a considerable amount of false alarms [Arp et al., 2022], always an undesirable paradigm for network operators.

Fortunately, the latest progress on programmable switches enables custom packet processing logic through domain-specific languages for the network data plane (e.g., the P4 language [Bosshart et al., 2014]). This leads to the possibility of potentially running complex ML models, such as DNNs, with high performance in the programmable data plane, thereby delegating complex decision-making to the fast path. In this thesis, we explore this possibility. But first, we give some background on ML, specifically DNNs.

2.4 Deep Neural Networks

Lately, Deep Neural Networks (DNNs) have been achieving unprecedented success in diverse fields of study [Xu et al., 2021, Xu et al., 2018]. With regard to NIDS, DL has already shown robust traffic classification [Boutaba et al., 2018]. However, implementing such algorithms on traditional PISA pipelines is difficult, if not impossible, since DL algorithms rely on linear algebra, requiring complex computations unavailable on PISA switches. Despite the existing barriers, prior art tried to map complex mathematical operations to the match/action tables by storing intermediate calculations [Zheng et al., 2022b]. Yet, as model size increases, the number of intermediate steps grows exponentially, limiting the size of the DL model.

Given the limitations of PISA architectures, new switch architectures and extensions to PISA have been proposed recently, potentially enabling running Deep Learning (DL) models [Swamy et al., 2022a] in the data plane, and therefore performing highly-accurate intrusion detection. In particular, recent server-based NIDS approaches using such models are showing promise by either reconstructing the network behavior through autoencoders [Mirsky et al., 2018] or extracting rich network features with Recurrent Neural Networks (RNNs) [Sohi et al., 2021]. We present details in these models next and leave the presentation of this new switch architecture to the next chapter.

2.4.1 Autoencoders

In general, as illustrated in Figure 2.3, the Autoencoder (AE) architecture aim to reconstruct a given input through DNNs, by encoding the input data into a lower dimensional latent space $\mathbf{z} \in \mathbb{R}^{d_\mathbf{z}}$ (Equation 2.1), followed by a decoder which maps the latent vector back to the input space $\mathbf{x} \in \mathbb{R}^{d_\mathbf{x}}$ (Equation 2.2).

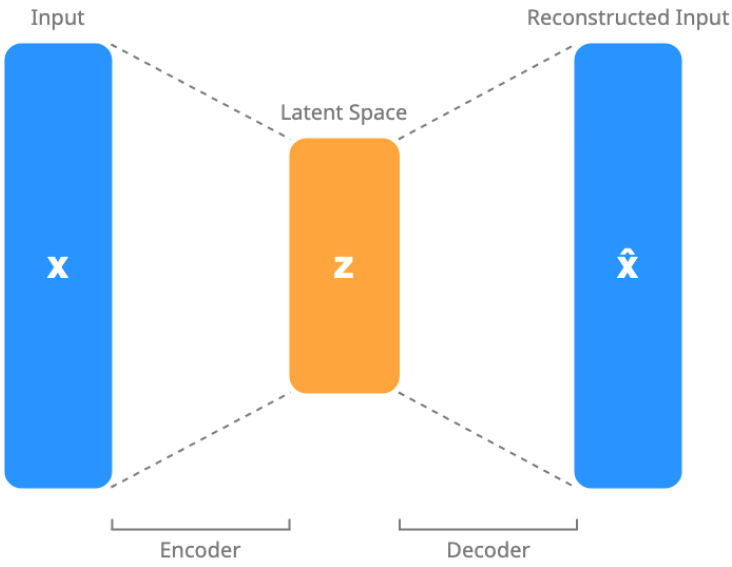

Figure 2.3: Traditional Autoencoder architecture

$$\mathbf{z} = g_e(\mathbf{W}_e\,\mathbf{x} + \mathbf{b}_e) \tag{2.1}$$

$$\hat{\mathbf{x}} = g_d(\mathbf{W}_d\,\mathbf{z} + \mathbf{b}_d) \tag{2.2}$$

Both equations 2.1 and 2.2 exhibit the computation of the encoder and decoder, respectively, with regard to its learned parameters $\mathbf{W}$ and $\mathbf{b}$, followed by an activation function g, typically a non-linear function (e.g., sigmoid or ReLU).

$$\mathcal{L}(\mathbf{x}, g_d(\mathbf{W}_d \; g_e(\mathbf{W}_e \, \mathbf{x} + \mathbf{b}_e) + \mathbf{b}_d)) = ||\mathbf{x} - \hat{\mathbf{x}}||_2^2 \tag{2.3}$$

In NIDS, since the learned parameters will be mainly centered around normal network patterns, being more difficult to reconstruct unseen abnormal behaviors, the anomaly score can therefore be measured as the reconstruction loss of the reconstructed input $\hat{\mathbf{x}} \in \mathbb{R}^{d_{\mathbf{x}}}$ with regard to the actual input $\mathbf{x}$ (Equation 2.3). Motivated by AEs, a state-of-the-art NIDS, Kitsune [Mirsky et al., 2018], implements an ensemble of autoencoder models capable of learning powerful distributions among the latent representations. Kitsune's ensemble consists of multiple AEs that together attempt to reconstruct various sets of features, grouped according to their similarity.

2.4.2 Variational Autoencoders

Variational Autoencoders (VAEs) are a type of neural network architecture that extends the concept of traditional Autoencoders (AEs) by introducing probabilistic modeling. VAEs are designed to encode input data into a lower-dimensional latent space while simultaneously learning the underlying probability distribution of data points in that latent space. This probabilistic approach allows VAEs to generate new data points that resemble the input data distribution.

Unlike in AEs, the encoder in the VAE takes an input data point and maps it to a probability distribution in the latent space. This is achieved through a DNN that encodes the input data into two vectors: a mean vector (μ) and a variance vector (σ^2). Then, a sampling step is introduced to generate a vector in the latent space from the learned distribution. The decoder in the VAE takes the sampled latent vector and maps it back to the data space to reconstruct the input data, exactly as done as in AEs. The key innovation of VAEs is the introduction of a loss function that combines two components: a reconstruction loss, as in AEs (Equation 2.3), and a regularization term based on the Kullback-Leibler (KL) divergence between the learned latent space distribution and a standard normal distribution. Given two continuous probability distribution q and p, the KL divergence is given by:

$$\mathcal{D}_{KL}(q||p) = \int_{\mathbf{x}} q(\mathbf{x}) \frac{p(\mathbf{x})}{q(\mathbf{x})} d\mathbf{x} \tag{2.4}$$

Thus, incorporating this probabilistic modeling into NIDS can enhance their effectiveness and adaptability in the evolving landscape of network security, offering a more comprehensive understanding of network behavior and improved detection capabilities.

2.4.3 Recurrent Neural Networks

Unlike other deep learning approaches, the Recurrent Neural Network (RNN) assumes that some data is sequential, such as text or time series data. On top of this assumption, RNNs are capable of extracting useful and powerful features for accurate inference. Thus, the input data $\mathbf{x}$ is decomposed in different t steps ($\mathbf{x}^1$, $\mathbf{x}^2$, ..., $\mathbf{x}^t$) and fed to each step of the RNN (Figure 2.4).

$$a^t = g_1(\mathbf{W}_a a^{t-1} + \mathbf{W}_\mathbf{x}\mathbf{x}^t + \mathbf{b}_a) \tag{2.5}$$

Similarly to traditional feed-forward neural networks, weight coefficients (i.e., $\mathbf{W}$ and $\mathbf{b}$) are taken into account. However, they will be shared temporally. At each step t, an activation vector a^t will be computed, considering the previous step activation and the input at its step (Equation 2.5). Finally, this activation can be reduced to produce an output value (Equation 2.6), e.g., an anomaly probability.

$$y^t = g_2(\mathbf{W}_y a^t + \mathbf{b}_y) \tag{2.6}$$

With respect to NIDS, RNNs can potentially lead to powerful solutions, considering the sequential flow patterns of networks that match well these models, motivating us to explore them in this work. However, RNNs are more computationally complex than most of the DL algorithms, making it even more challenging to fit the resource constraints of a network switch.

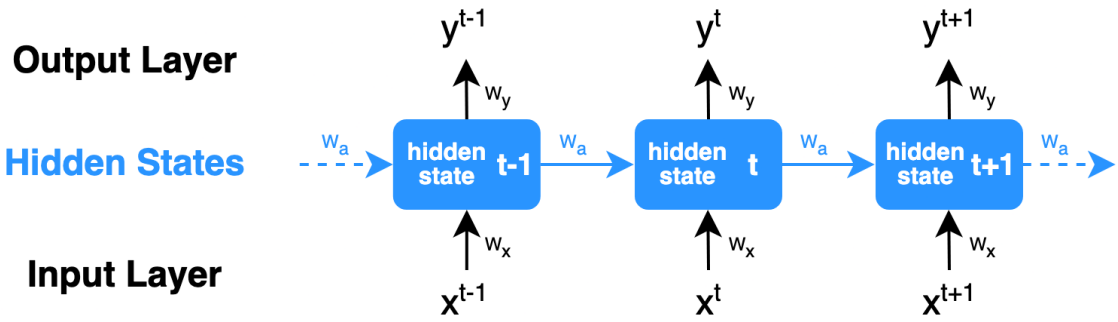

Figure 2.4: A traditional Recurrent Neural Network (RNN)

2.4.4 Gated Mechanisms

Some variants of RNNs, namely the Long Short-Term Memory (LSTM) [Hochreiter and Schmidhuber, 1997] and the Gated Recurrent Unit (GRU) [Chung et al., 2014], propose modifications to the RNN architecture in order to solve the vanishing gradient problem encountered by traditional RNNs. Because

of the nature of simple RNNs, this problem is encountered because the information is lost through time and the model is not able to encode all the information inside the timestep hidden vector. So, the main change in both LSTMs and GRUs is that they implement gate mechanisms that allow them to better regulate the flow of information over time. Essentially, each of the network cells will decide how much information will be carried to the next memory cell, thus dropping information if it is irrelevant to the task, or maintaining it if so. GRUs implement this method by introducing two specific gates.

Reset Gate

The reset signal r^t (Equation 2.8) is responsible for determining how important h^{t-1} is to the summarization $\tilde{h}^t$. The reset gate has the ability to completely ignore the previous hidden state if it turns out that h^{t-1} is irrelevant to the computation of the new memory (Equation 2.9).

Update Gate

The update signal z^t (Equation 2.7) is in charge of determining how much of the previous hidden state h^{t-1} should be carried forward to the current state h^t. For example, if $z^t \approx 1$, then h^{t-1} is almost entirely copied out to h^t . In contrast, if $z^t \approx 0$, then mostly the new memory $\tilde{h}^t$ is forwarded to the current hidden state. Equation 2.10 is illustrated by this example.

$$z^t = g(\mathbf{W}_z \mathbf{x}^t + \mathbf{U}_z h^{t-1}) \tag{2.7}$$

$$r^t = g(\mathbf{W}_r x^t + \mathbf{U}_r h^{t-1}) \tag{2.8}$$

$$\tilde{h}^t = g(r^t * \mathbf{U}_x h^{t-1} + \mathbf{W}_x \mathbf{x}^t) \tag{2.9}$$

$$h^t = (1 - z^t) * \tilde{h}^t + z^t * h^{t-1} \tag{2.10}$$

Because the LSTM introduces additional gates, its number of parameters turns out to be significantly greater, losing on inference speed as its complexity increases. This makes LSTMs slightly less appealing to use in the context of NIDS, since it may affect throughput if one runs it on a per-packet basis. Nevertheless, in general, RNNs can provide richer features to the NIDS that specify the flow patterns of the network, not being limited to the flow information of the feature extractor. Consequently, gated mechanisms may better recognize when certain patterns become irrelevant or pertinent at a specific time, only maintaining the most important encoded information required by the classification model. So there is an efficiency-performance trade-off worth exploring.

2.4.5 Deep Gaussian Models

Deep learning is a hierarchical inference method composed of one or more sequential layers able to efficiently describe complex relationships, by working on feature combinations. In parallel, Gaussian-based models can represent data within multivariate standard Gaussian distributions which follow certain prior probabilities to classify data points [Reynolds et al., 2009]. A combination of Deep Learning with Gaussian methods would essentially benefit from the best of both worlds (Figure 2.5). The Deep Gaussian Mixture Model (DGMM) [Viroli and McLachlan, 2019] introduces a network of multiple layers of latent variables that follow a mixture of Gaussian distributions. The gains from integrating deep networks on Gaussian methods are: 1) benefit from reduced data dimensionality, therefore reducing the complexity of the model; 2) enhance the quality of features through the Deep Learning (DL) extrapolation, allowing a more robust representation of data. As a result, the combination of these techniques can be highly advantageous in the context of NIDS, where the ability to deduce Gaussian probabilities from improved prior intrusion distributions can significantly enhance the system's efficiency.

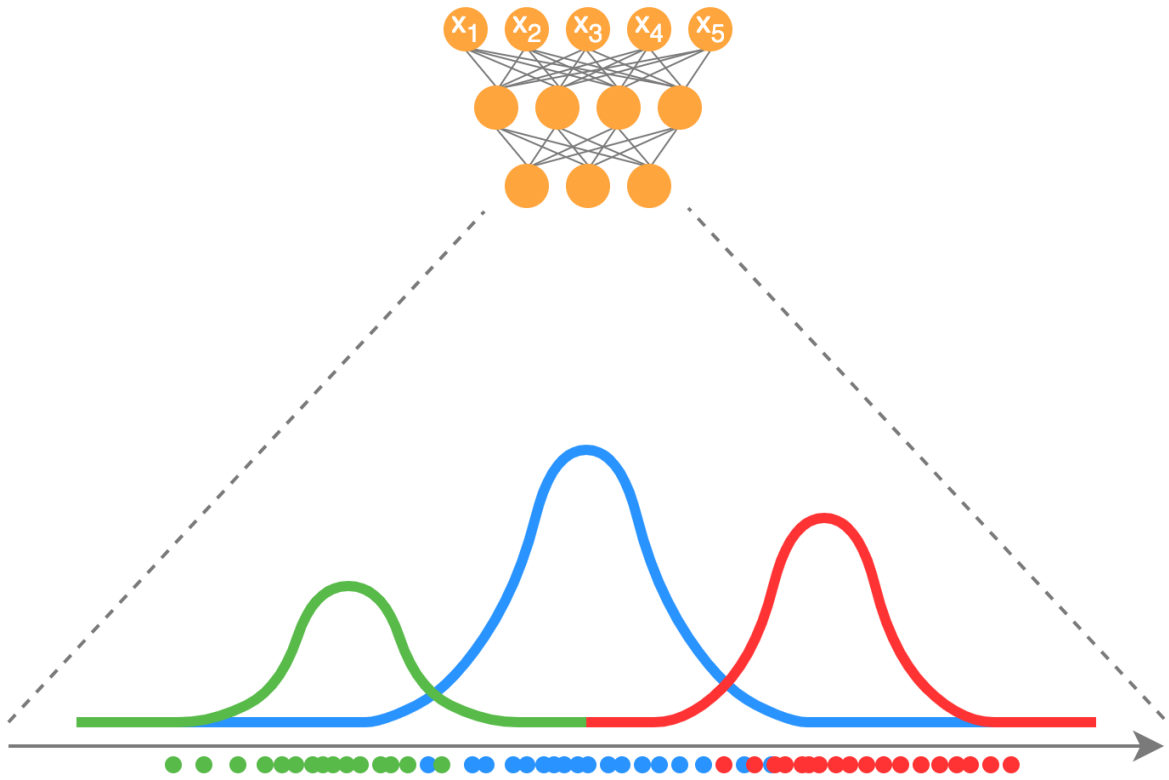

Figure 2.5: Latent vector mapped into a Gaussian Mixture Model (GMM)

2.5 Techniques to enable ML on constrained devices

The selection of DNN models we made to explore in this thesis is grounded in our goal of achieving good detection performance while respecting the resource constraints of network switch architectures. In addition, we will also explore techniques that can further reduce the computational requirements of these models to allow their practical deployment.

2.5.1 Feature Importance

Computing feature importance enables successful feature selection to reduce the dataset's dimensionality by choosing the most relevant features to a given task. Therefore, feature importance calculation has become a key data analysis tool to better study a wide range of datasets. Such methods can go from simple permutations of features to even more complex entropy computations. However, both can provide insightful interpretations of features.

The permutation feature importance measurement was first introduced by Random Forests (RFs) [Breiman, 2001]. In general, permutation feature importance gauges the variation in the prediction error of the model after the permutation of the feature values, which typically will break the relationship between the feature and the true outcome. Using this method, a feature can be considered as important if noising its values increases the model uncertainty because in this scenario the model relied on the feature for the prediction. On the other hand, if noising its values does not affect the model predictions, it can be considered that the model ignored the feature for the prediction, and this feature is most probably irrelevant.

Similarly to Random Forests (RFs), XGBoost (XGB) [Chen and Guestrin, 2016], a gradient boosted decision tree system, employs feature importance methods in order to succeed. Typically, feature selection in gradient boosted decision trees comes from the information gain regarding the selection of a particular feature. Equation 2.11 presents the calculation of the information gained, which is often regarded as the degree of uncertainty (i.e., entropy) difference between the current node and the subsequent leaves. For 2-class classification, the entropy would be calculated as shown in Equation 2.12, where P and N stand for the ratio of positive and negative data points respectively. In general, feature importance methods calculate how well a given feature splits the dataset, according to the ground truth. In case a feature is capable of separating all the data points within the dataset, it is an indicator of its high information gain.

$$gain(f) = entropy(node) – [average\ entropy(leaves)] \tag{2.11}$$

$$entropy(P, N) = -P \log_2(P) - N \log_2(N) \tag{2.12}$$

Thus, for NIDS, feature importance is a relevant area of study, since these ML systems can benefit from a better selection of features, fulfilling a lighter representation learning. In addition, it provides an enhanced comprehension of the dataset features, perhaps making the ML model less confused, minimizing entropy, and preventing it from weak assumptions.

2.5.2 Quantization

Despite the effectiveness of deep neural networks, it is clear that their high computational costs are still a barrier to being able to perform their inference without dedicated hardware. Though, in a broad range of tasks, to achieve state-of-the-art results, DNNs are increasingly larger. In the case of programmable switches, it is evident that these network devices lack the necessary resources to run the complex arithmetic of DNNs. One can however leverage DNNs quantization, an active research topic, in order to effectively find lower bit-width model representations rather than executing a costly full-precision representation. Quantization is, therefore, a mechanism to reduce the computational and memory costs of running inference by representing the weights and activations with low-precision data types such as 8-bit integer (int8), or even lower integer scales, instead of the conventional 32-bit floating point (float32). Reducing the number of bits means the resulting model requires less memory storage, theoretically consumes less energy, and linear algebra operations. For instance, matrix multiplication can be performed much faster with integer arithmetic. It also enables running such models on some resource-constraint devices, which sometimes only support integer data types.

Integer Quantization

Generally, converting floating-point numbers to fixed-point scales requires efficient ways of mapping those numbers across scales. Eventually, the quantization algorithm will certainly incur some loss, since it is inevitable that a coarse-grained scale will contain several fine-grained values within the same interval. For that reason, an integer could for instance represent various floating-point values at the same time, degrading the model precision. Naturally, quantization can be assumed as an approximation method that accelerates the model execution. Thus, the less number of bits used to quantize a model, the more loss the quantization algorithm may be subject to, and consequently better the inference performance. The simple range-based quantization is a great example to illustrate how quantization works. As Figure 2.6 suggests, range-based quantization simply would map float32 values to the int8 representation. In this case, quantization is asymmetric, which means the zero-point in both scales is not aligned, and so the quantization process requires the addition of the distance between both zero-points, which effectively represents the origin zero-point shift.

As presented in Equation 2.13, the quantized integer representation of a floating-point number can be achieved by computing two linear transformations: applying scale and the zero-point (also known as quantization bias, or offset).

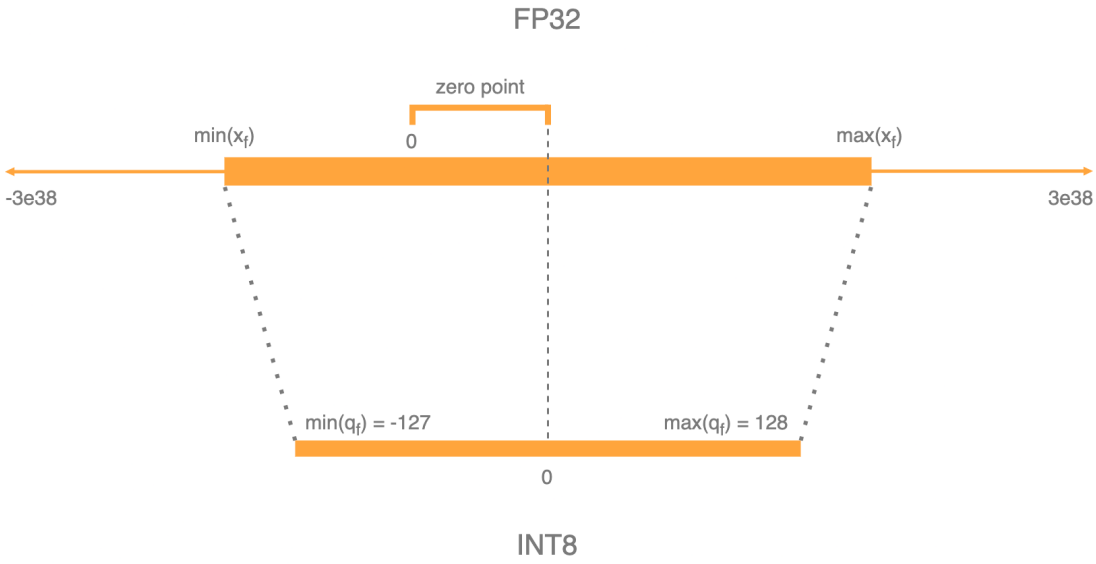

Figure 2.6: Assymmetric range-based integer quantization

$$Q(x_f) = round(\frac{x}{scale} + zero\ point) \tag{2.13}$$

$$scale = \frac{max(x_f) - min(x_f)}{max(q_f) - min(q_f)} \tag{2.14}$$

$$zero\ point = \frac{max(x_f) + min(x_f)}{2} \tag{2.15}$$

Some quantization algorithms can work with as low as 2-bit or even 1-bit integer precisions. For instance, Binary Neural Networks (BNN) is a DNNs quantization method, which represents each neuron as a 1-bit integer (e.g., the sign of the value). BNNs were actually experimented in the programmable data plane [Siracusano and Bifulco, 2018,Zheng et al., 2022b], however being very imprecise and requiring tens of match/action tables. Further, interesting work has been proposed in the fields of computer vision and Large Language Models (LLMs), where models tend to be particularly exhaustive, bringing complex applications to portable devices. For that purpose, Yang et. al [Yang et al., 2019] formulated the quantization operation as a differentiable non-linear function for training aware multi-bit quantization of both weights and activations. Further, it is also possible to quantize full-precision networks into low-bit networks through hashing mechanisms [Cao et al., 2017]. Nevertheless, quantization is a vast area of research, and there are multiple complex methods to approach the quantization problem. In

general, quantization methods fall broadly into two categories: **Post-Training Quantization (PTQ)** and **Quantization Aware Training (QAT)**.

Post-Training Quantization

Post-training methods of quantization generally involve quantizing a model after it has been trained. Therefore, the model has already acquired the general knowledge about the given task, and quantization takes place to convert the learned parameters to a lower bit-width. In some cases, in order to soften the quantization error, a portion of the dataset is chosen to perform some training steps (i.e., finetuning), which requires much less computation when compared to full model training, so that the model can adapt to the change of scale. Post-training approaches are therefore particularly appealing for larger models, for which full training is expensive. In addition, some methods can quantize components to 3-4 bits PTQ methods [Frantar et al., 2023], while remaining accurate. However, this kind of approach usually targets models with hundreds of billions of parameters (e.g., LLMs), which most of the time are overparametrized networks that can be subject to high degrees of compression without suffering high degradation.

Quantization Aware Training

The other class of quantization methods is Quantization Aware Training (QAT). Unlike the PTQ methods, QAT algorithms typically quantize models during extensive retraining or full training, using special differentiation mechanisms to adjust the quantized weights and activations during the training phase [Gholami et al., 2021, Nagel et al., 2021]. This training optimizes the quantized model weights to enhance model performance on the given task by emulating inference-time quantization. It uses ”fake” quantization blocks, known as quantization and dequantization blocks, during the training phase to reproduce the behavior of the inference phase. In sum, these blocks are introduced to serve as gates that will transform the original precision to the quantized lower bit-width precision, in order to differentiate the quantization operation during training. Figure 2.7 represents the differentiable relations within the quantized model execution. For smaller models, which usually do not require large amounts of resources to perform a full training phase (i.e., exposing the quantized model to the entire set of training data), QAT can be very beneficial since the knowledge can be better transmitted to the quantized weights. In contrast, PTQ methods would only expose the quantized parameters to a very restricted portion of the training data, therefore not performing so well for smaller networks.

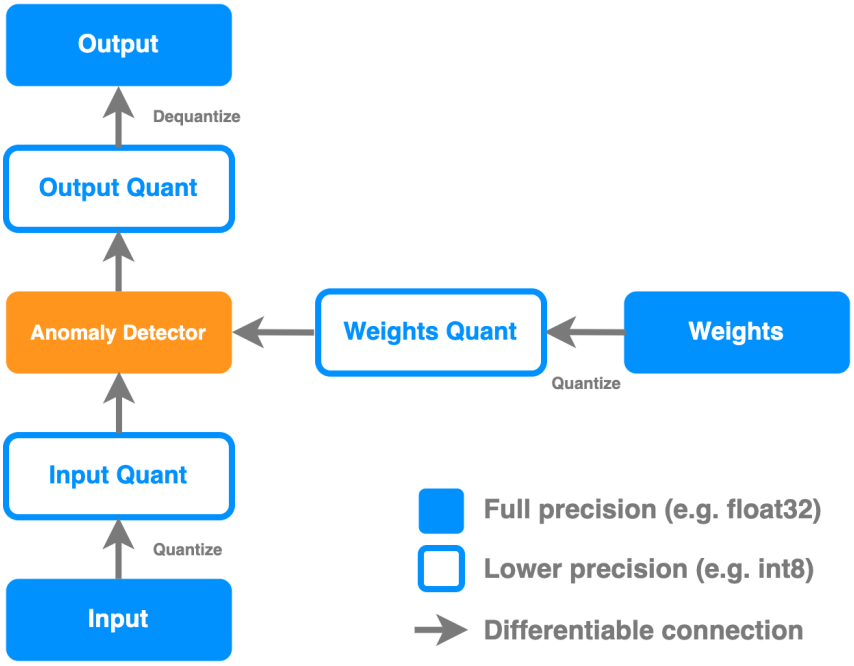

Figure 2.7: Quantize and Dequantize operations in QAT

2.6 Summary

In this background discussion, we mainly explored the potential of Deep Neural Network (DNN) models in the context of Network Intrusion Detection Systems (NIDS) as well as useful techniques that may help compress such models' execution. By incorporating these methods, NIDS can more efficiently navigate the complex and dynamic world of network security, resulting in more effective threat detection and a deeper understanding of network behavior. In addition, we also compared these ML-based approaches with existing signature-based solutions for intrusion detection.

Chapter 3

Related Work

In this chapter, we explore related work and discuss it from multiple perspectives. This chapter is dedicated to establishing a comprehensive context for the current state-of-the-art in the application of ML within the networking domain, placing special emphasis on our central challenge: the detection of network anomalies.

3.1 ML in Networking

The recent advances in ML led various learning-based models to enable powerful solutions to many worldwide problems, including networking systems, due to their intrinsic capability of extracting knowledge from data and identifying crucial *hidden* patterns. The application of ML for traffic classification has been a research topic for about two decades now [Zuev and Moore, 2005]. In the last few years, many new use cases have been emerging in networking systems, including traffic routing [Medhi and Ramasamy, 2017], congestion control [Alcoz et al., 2022] and intrusion detection [Mirsky et al., 2018]. As a result, ML started outperforming previous non-ML handwritten heuristics in all these fields [Boutaba et al., 2018].

Running the complex ML computation within network devices was unfeasible until recently, and all ML-based networking solutions were thus deployed in server-based middleboxes. But key technological progress in networking empowered fast and programmable network devices (e.g., switch ASICs [Intel, 2016] and SmartNICs [Firestone et al., 2018]), progressively enabling the use of ML in the network data plane. A recent trend is thus the acceleration of networking ML tasks, resorting to programmable switches [Sapio et al., 2019, Swamy et al., 2022a, Amado et al., 2023].

3.2 The Challenge of ML-based Anomaly Detection

The efforts made building innovative anomaly detectors have been increasing significantly over the past few years, owing much to the outstanding success of Deep Learning. In particular, many areas can benefit from anomaly detection, primarily due to its ability to extrapolate data from limited knowledge, identifying anomalous segments in the feature space. Grasping the abilities of unsupervised learning, Xu et al. [Xu et al., 2018] proposed a reconstruction-based anomaly detector for seasonal Key Performance Indicators (KPIs), based on a Variational Auto-encoder (VAE). Still, motivated by the emergence of Transformers [Vaswani et al., 2017], Xu et al. [Xu et al., 2021] advanced the state-of-the-art in Time-Series anomaly detection. While applied to other fields, these results also show promise to bring such approaches into the networking ML context. However, they may require either heavy compression or quantization methods, as a consequence of the compute restrictions of existing programmable switching pipelines.

In general, the use of deep learning is not something new in the field of network intrusion detection. Accordingly, Kitsune [Mirsky et al., 2018] proposed a deep learning approach for NIDS, leveraged by the success of unsupervised learning. By using an ensemble of auto-encoders, which will attempt to reconstruct the network traffic patterns from the extracted features, the authors were able to build a generic framework for network intrusion detection. The main assumption behind Kitsune is that, while the model will easily reconstruct benign traffic through the auto-encoder, it will actually present a higher reconstruction loss whenever malicious traffic is passing through the network, due to the fact that it was only trained with benign instances. However, since neural networks are computationally expensive, this and other proposed art using deep learning (e.g., [Tang et al., 2016]) are still unfeasible to deploy to current network data planes, requiring a new switch architecture [Swamy et al., 2022a] to enable in-network per-packet ML, or either offloading the model to the control plane, with a significant increase in overhead, which is the common case.

More recently, Fu et al. [Fu et al., 2021] presented a malicious traffic detector, that employs frequency domain analysis, increasing accuracy and robustness. In this work, the per-packet features are encoded as a matrix composed of a set of feature vectors of each packet. A linear transformation is applied to this matrix to reduce the dimensionality of the problem, without information loss. Finally, Discrete Fourier Transformation and Logarithmic Transformation are applied before feeding the extracted flow patterns to a pretrained clustering module, based on the K-Means clustering algorithm. While achieving higher performances (tens of *gbps*) than the previous state-of-the-art (Kitsune [Mirsky et al., 2018]), it is still a server-based solution, and therefore not able to achieve line-rate. It is also unclear if the complex computations involved in feature computation and ML inference are feasible to deploy in network switches data planes any time soon.

3.3 Deploying ML Models in the Network Data Plane

The research community has already started exploring the possibility of running ML inference in the network data plane, to achieve line-rate processing. One of the earlier examples was the introduction of Decision Trees (DT). In this case, each stage of the match/action pipeline can represent one level of the tree (Figure 3.1), while the packets convey the feature values [Coralie et al., 2019]. The main problem of this approach is the very limited depth of the tree, reducing its applicability. To address this, new model encoding mechanisms were proposed, boosting the efficiency, by using fewer pipeline stages and enabling larger trees [Zheng et al., 2022a, Zheng et al., 2022b, Zhou et al., 2023]. Further, benefiting from the improved encodings, IIsy [Zheng et al., 2022a], Planter [Zheng et al., 2022b] and NetBeacon [Zhou et al., 2023] proposed advanced DT-based models in the programmable data plane, specifically, Random Forests (RFs) [Breiman, 2001] and XGBoost (XGB) [Chen and Guestrin, 2016]. In addition, Mousika [Xie et al., 2022] uses Binary Decision Trees (BDT), designed with the particular purpose of ML inference in programmable switching pipelines. Typically, BDT sniffs the bits of each numeric feature (i.e., binary numbers), by decomposing each trained DT branch into multiple ones. In fact, a straightforward conversion of DT to BDT would face a combinatorial explosion issue with an increase of embedded features in a DT [Zhou et al., 2023]. However, Mousika trains its BDT model with binary features, softening this potential issue. Although these methods are able to deploy per-packet line-rate anomaly detectors on the programmable data plane, they still are weak in terms of detection performance, due to the intrinsic limitations of the ML models for this task.

Other types of methods have also been introduced. Recently, Alcoz et al. [Alcoz et al., 2022] proposed a fully automated per-packet detection system specifically targeting pulse-wave DDoS attacks, based on an Aggregate-Based Congestion Control mechanism. The main achievement of this work was its fast reaction times, crucial to detecting series of short-duration high-rate traffic pulses. However, such efficiency depends on the quality of the retrieved features, so an online-clustering module is used to map malicious traffic as evident outliers. Further, the statistics of the cluster are computed by the controller, and a programmable scheduler takes action, by mitigating possible anomalous packets, allowing the benign traffic to proceed fluidly. In this case, therefore, the ML task is not running in the data plane.

3.4 Per-Packet ML: Limitations & Opportunities

3.4.1 Limitations

The integration of ML mechanisms in the network data plane can definitely establish new opportunities, by enabling intelligent traffic analysis at line-speed rates. However, despite the advantages of data plane ML, it is crucial to consider the existing limitations of current programmable data planes to understand if

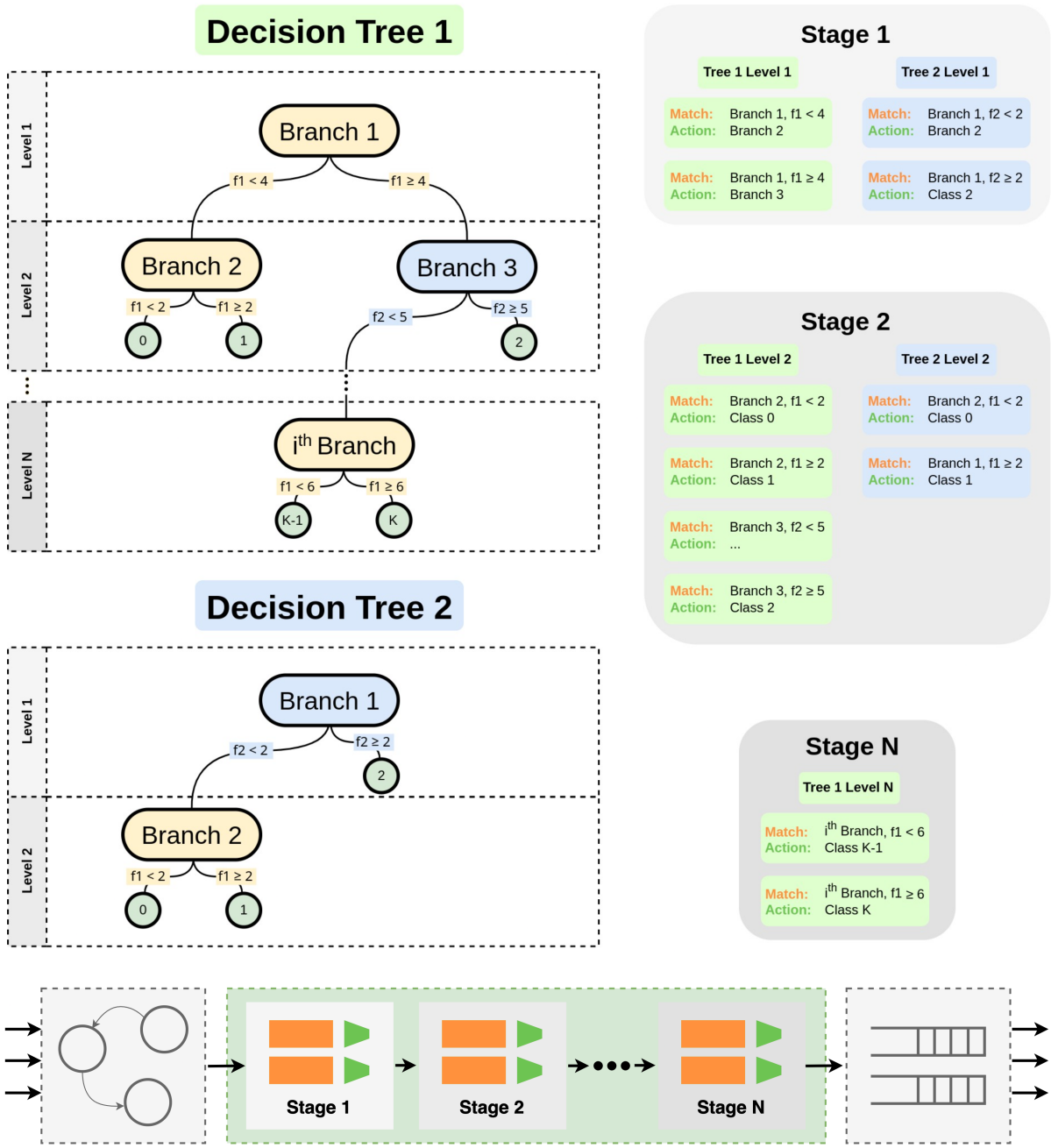

Figure 3.1: DT-based ensemble model directly mapped to the Protocol Independent Switch Architecture (PISA). Each level of each decision tree is transformed into match/action rules for further inference, within its respective pipeline stage.

the necessary trade-offs are favorable. For instance, arithmetic operations are very limited: multiplication and division are not supported, as well as floating point operations. In addition, even simple operations have to be mapped to the PISA's match/action pipeline, which turns out to be composed of a limited number of stages with limited access to stateful memory. Therefore, implementing ML models in the data plane is hard, yet only possible for very simple models, but it seems strictly impossible for more

complex models, such as the DNNs that have shown promise in the detection of malicious traffic.

3.4.2 Opportunity: A New Per-Packet ML Architecture

Current ML models that run on top of PISA-based data planes are still far from achieving the outstanding results of their server-based counterparts. Unfortunately, current match/action pipelines cannot apply complex techniques (e.g., deep learning models) due to their extensive representation [Fu et al., 2021], raising their value by capturing richer and more useful features. Thus, there is a need for either extending PISA with ML-friendly primitives, or designing innovative switch architectures that can mitigate the aforementioned limitations, leveraging useful capabilities that may benefit complex decision-making, while maintaining the required *tbps* scales. One recent alternative is Coarse-Grained Reconfigurable Arrays (CGRAs), which have gained popularity as these are optimized for ML arithmetic, by capturing effective data locality and memory access patterns. So, looking to provide high computational throughput, Taurus [Swamy et al., 2022a] proposed an efficient data plane architecture, which extends PISA with a parallel-patterns abstraction, based on Plasticine [Prabhakar et al., 2017], a CGRA composed of multiple compute and memory units, replacing some of the MATs by a *MapReduce* block (Figure 3.2).

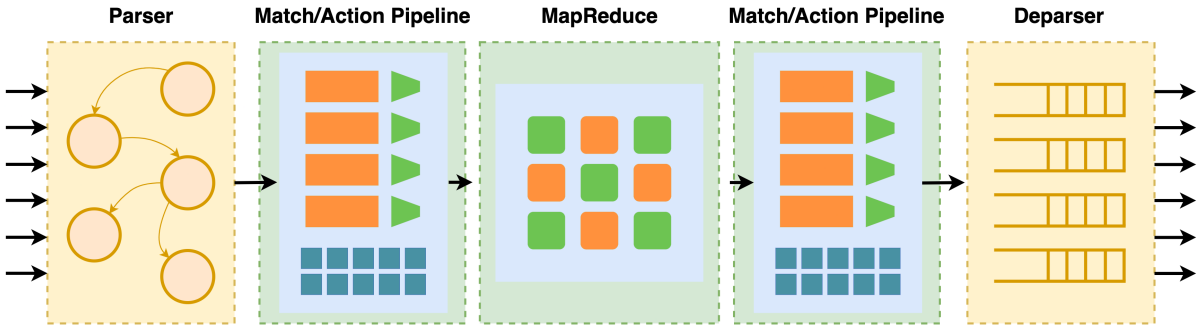

Figure 3.2: Taurus architecture [Swamy et al., 2022a]

The success of Taurus comes therefore from the use of the pipelined Single Instruction Multiple Data (SIMD) parallelism, unlike the Very Long Instruction Word (VLIW) architectures commonly used in the current programmable switches. In particular, Machine Learning (ML) computations are easily vectorized, ending up consisting of repeated mathematical operations (e.g., matrix-vector multiplications). This profile of computation is then made more suitable for SIMD parallelism. As a result, the Taurus architecture is able to perform per-packet ML more efficiently than match/action pipelines, bringing complex decision-making to the data plane (ensuring line-rate computing with nano-second latencies). Particularly, the authors of Taurus explored their proposed architecture by developing a small anomaly detection prototype, using a 4-layer Deep Neural Network (DNN), based on the work proposed by Tang et al. [Tang et al., 2016]. In this thesis, we used Taurus to investigate different types of DNNs in NIDS

and to study the trade-offs of using approximation techniques to fit these models in constrained devices.

The Taurus core logic to support these arithmetic operations is in the *MapReduce* block, which can be configured by a Domain Specific Language (DSL) for application accelerators (i.e., Spatial [Koeplinger et al., 2018]). This block organizes vector data on multiple threads, as known as Functional Units (FUs), that will perform one *MapReduce* operation. The result of this computation is then passed to a Pipeline Register (PR) that will maintain the intermediate result along cycles. This described pipeline (Figure 3.3) composes one Taurus Compute Unit (CU). Taurus contains as many of these CUs as needed to execute the ML model, at the cost of losing throughput and consuming more power. Since there is no ongoing production of the Taurus chip, this *MapReduce* abstraction was deployed in a Field Programmable Gate Array (FPGA) [Swamy et al., 2022a]. In this thesis, we explore the potential of Taurus for DNN-based malicious network detection.

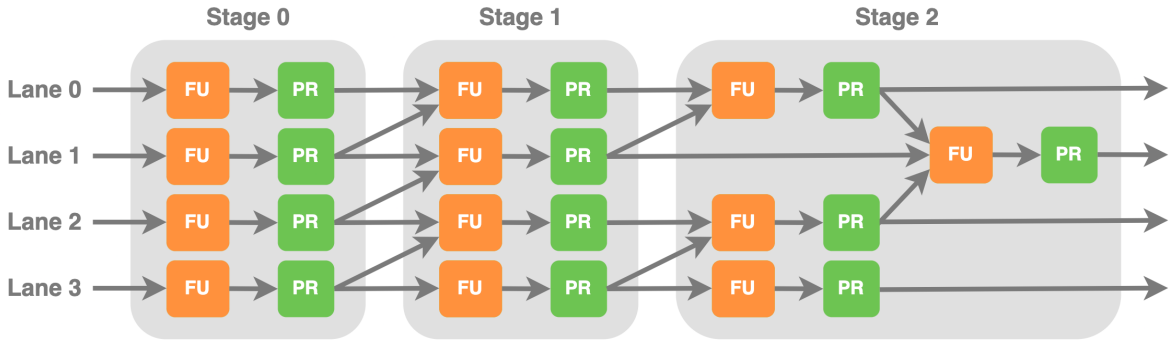

Figure 3.3: A three-stage CU, composed of Functional Units (FUs) and Pipeline Registers (PRs) [Swamy et al., 2022a].

3.5 ML Feature Computation in the Network Data Plane

While so far we have considered the computational difficulty of running ML in the data plane, we have missed another challenge: computing the features that feed the ML model. In fact, this component of the ML pipeline is often the most resource-consuming [Mirsky et al., 2018, Fu et al., 2021].

The emergence of programmable switches and the flexibility of PISA, not only inspired the research community to perform ML inference at the data plane, as well as computing the useful features that feed the deployed models. This module for feature computation extrapolates the problem of running an entire ML pipeline in the data plane, resulting in more hardware consumption, together with potential increased overhead.

Recent work has proposed new traffic analysis mechanisms in the network data plane, with the goal of collecting per-packet and flow-level features at line-rate. While the former are stateless features from

individual packets (e.g., typically fetched from the packet headers), the latter generally relies on stateful statistics from the packets with regard to the respective flow (e.g., inter-packet delay). However, flow-level features are often based on aggregations, such as mean and variance, which requires operations not supported by PISA switches (e.g., multiplications and divisions).

FlowLens [Barradas et al., 2021] extracts advanced flow-level counters on the data plane (e.g. packet size), but performs flow-level feature computation in the control plane, thereby incurring in additional overhead. Poseidon [Zhang et al., 2020] and Jaqen [Liu et al., 2021], two misuse-based intrusion detectors, extract some stateful flow information at line-rate in the data plane, though with occasional control plane interventions. On the other hand, Planter [Zheng et al., 2022b] and Mousika [Xie et al., 2022] only extract stateless per-packet features in the data plane, though with strict limitations and fully ignoring useful flow information, crucial for robust detection. Finally, NetBeacon [Zhou et al., 2023] combines per-packet and flow-level features, successfully boosting traffic analysis effectiveness. In particular, NetBeacon computes these flow-level features at the data-plane level by using approximation methods. More recently, Peregrine [Amado et al., 2023], a line-rate anomaly-based NIDS, performs feature computation entirely in the data plane, while the ML inference module receives the computed features in the network control plane at every *epoch*, to perform anomaly detection. We plan to leverage the feature computation module from Peregrine and integrate it with an in-network ML module running also in the data plane.

3.6 Summary

We present a summary table (see Table 3.1) to recap the discussed state-of-the-art and compare it with the approach we proposed in this thesis. Our solution is the only one that performs flow-level feature computation and DNN-based inference entirely in the network data plane. In addition, we introduce experiments using distinct DNN models and powerful techniques to compress such neural networks.

NIDS	Features	Line-Rate	DP FC	DP ML	DNNs
Kitsune [Mirsky et al., 2018]	Flow-Level	✗	✗	✗	✓
pForest [Coralie et al., 2019]	Per-Packet	✓	✓	✓	✗
ACC-Turbo [Alcoz et al., 2022]	Flow-Level	✓	✗	✗	✗
Planter [Zheng et al., 2022b]	Per-Packet	✓	✓	✓	✗
IIsy [Zheng et al., 2022a]	Flow-Level	✓	✓	✓	✗
Mousika [Xie et al., 2022]	Per-Packet	✓	✓	✓	✗
Whisper [Fu et al., 2021]	Flow-level	✗	✗	✗	✗
NetBeacon [Zhou et al., 2023]	Flow-level	✓	✓	✓	✗
Peregrine [Amado et al., 2023]	Flow-level	✓	✓	✗	✓
Our Work	Flow-level	✓	✓	✓	✓

Table 3.1: ML-powered NIDS solutions comparison

Chapter 4

Proposed Approach

In this thesis, our goal is to evaluate different techniques to improve the efficiency of ML models under resource constraints, with a focus on network-based anomaly detection. In order to be one step closer to running complex DNN models in the limited network data plane, we explore a concise and powerful representation learning to decrease the processing latency of such models and achieve the desirable line-rate performance. On top of that, we resort to multiple methods to compress the execution of ML models. We also explore new switching hardware that has the primitives and the resources to execute more complex models.

4.1 Representation Learning

In theory, running complex ML model in network devices requires lightweight model representations, in order to allow such models to be successfully deployed. However, modern networks require ML-based anomaly detectors that are able to capture complex interactions within the environment, for instance by extrapolating the original extracted metadata regarding packet and flow statistics.

4.1.1 Model #1: Autoencoder

One of the models we investigate in this thesis is the Autoencoder (AE), recently shown to be a successful model for NIDS with the Kitsune [Mirsky et al., 2018] anomaly detector, which is based on an ensemble of AEs. Contrary to Kitsune, where features are mapped to different AEs in the ensemble, our approach removes the mapping computation while only having one single AE. By using only one AE, instead of an ensemble, the anomaly detector turns out to be significantly more efficient since features can be directly reconstructed in parallel (in the next chapter we compare some results with Kitsune). In contrast with our approach, Kitsune introduces an additional output AE layer on top of the ensemble AE

layer, which will increase inference overhead since it has to be performed sequentially. Instead, we propose a simple lightweight autoencoder supported by a wise selection of features, and show it is capable of achieving competitive detection results. Figure 4.1 presents the architecture of Kitsune and the one we investigate in our thesis. Next, we detail our architecture components.

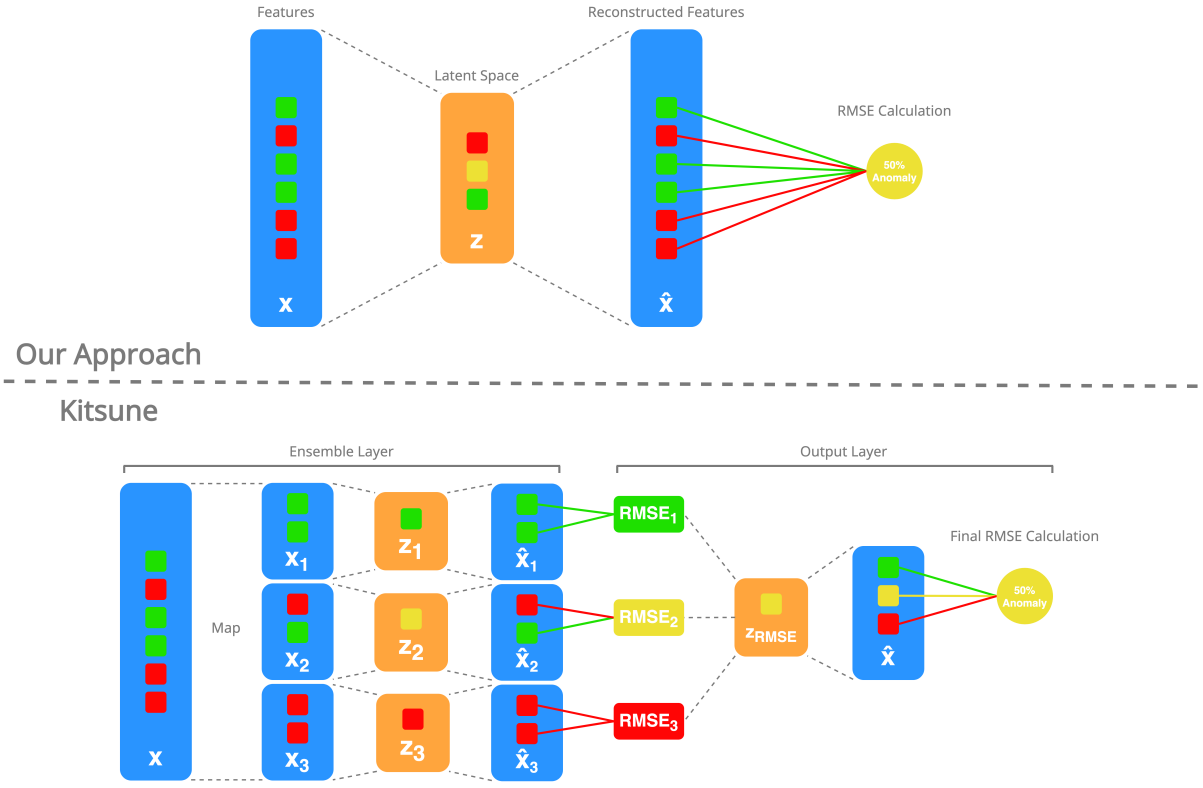

Figure 4.1: Our AE model vs. Kitsune [Mirsky et al., 2018]

1. Input Layer

Each selected feature is connected to each neuron of the input layer, being $\mathcal{S}$ the number of selected features during the training phase: $\mathbf{X} = (\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_\mathcal{S})$. The smaller the number of selected features, the more efficient the inference computation will be.

2. Latent Vector

The vector containing the selected features is then embedded into the encoder to be transformed into a latent vector (Equation 2.2). This non-linear transformation extrapolates a low-dimensional representation space, that will force the model to perform a clear distinction between normal and anomalous packets. In our thesis, we explore different non-linear activation functions, evaluating their computational impact on the overall system.

3. Reconstruction Layer

The transformed vector will then be presented as input to the decoder that will be in charge of mapping the latent vector back to the input dimensional space. The assumption is that the model, during its training phase, will only be exposed to the normal behavior of the network. For that reason, it will learn enriched representations of normal network flows. In contrast, during its testing, the model can be subject to abnormal occurrences in the network, and because it was only trained using non-anomalous packets, it will not be able to precisely reconstruct patterns representing anomalies. Given this assumption, we can formulate the loss function as the Root Mean Squared Error (RMSE) between the input and reconstructed features (greater RMSE, greater the anomaly probability):

$$\text{RMSE} = \sqrt{\frac{1}{\mathcal{S}}\sum_{i=1}^{\mathcal{S}}(\mathbf{x}_i - \hat{\mathbf{x}}_i)^2} \tag{4.1}$$

4.1.2 Model #2: Gated Recurrent Autoencoder

Most attacks tend to have some sequentiality, as packets from the same flow contain relevant sequential information to the detection task. This means that features representing the packets regarding the same flow can benefit detection performance. In particular, RNN-based models have shown superiority in NIDS when compared to non-sequential models [Sohi et al., 2021]. Although flow statistics can already provide enhanced results (i.e., rather than just relying on individual packet features) [Mirsky et al., 2018, Fu et al., 2021], there is still the need to go even further and carry a vector representation of the network flows. Thus, GRUs, a variation of RNNs to capture long-term dependencies, can supply enhanced sequential information and additionally filter out the relevant information for intrusion detection from unimportant encodings. One concern regarding RNNs are often discarded due to their long processing latency, however wise mechanisms have recently allowed running these models with fewer computational resources by storing the previous GRU state into cache [Branco et al., 2020]. Therefore, based on the success of GRUs and the importance of extrapolating richer representations of the network flows, we propose to investigate a Gated Recurrent Autoencoder for our anomaly detection task. The approach is as follows:

1. Flow Representation

Following the same logic of Peregrine's feature extractor [Amado et al., 2023], we use the packet's *5-tuple* (i.e., composed of source and destination IP, source and destination port, and the protocol) as the flow key to accurately identify each flow in the network. As presented in Figure 4.2, every flow will be assigned with an initial GRU hidden state. This initial GRU state can be learned and it is basically the default GRU state for every flow. However, as new packets for a flow arrive, new states are computed, aiding the reconstruction process of a packet. After inference, the previous state will be discarded,

whereas the current one is saved into the cache, so the next arriving packet for the given flow uses this more recent state for inference. Using the GRU gates, the model will naturally promote or diminish information, based on how relevant it is for classification, so the next packet anomaly score can be precisely computed.

2. Recurrent Reconstruction

The reconstruction process in the Gated Recurrent Autoencoder is the same as in the Autoencoder, but instead leverages the sequential encoded states. The calculated GRU state will be used as the latent representation of the packet, and it will pass through the decoder layer in order to be transformed back to the input's dimension. Finally, the reconstruction error function is also the same as Autoencoder's (Equation 4.1).

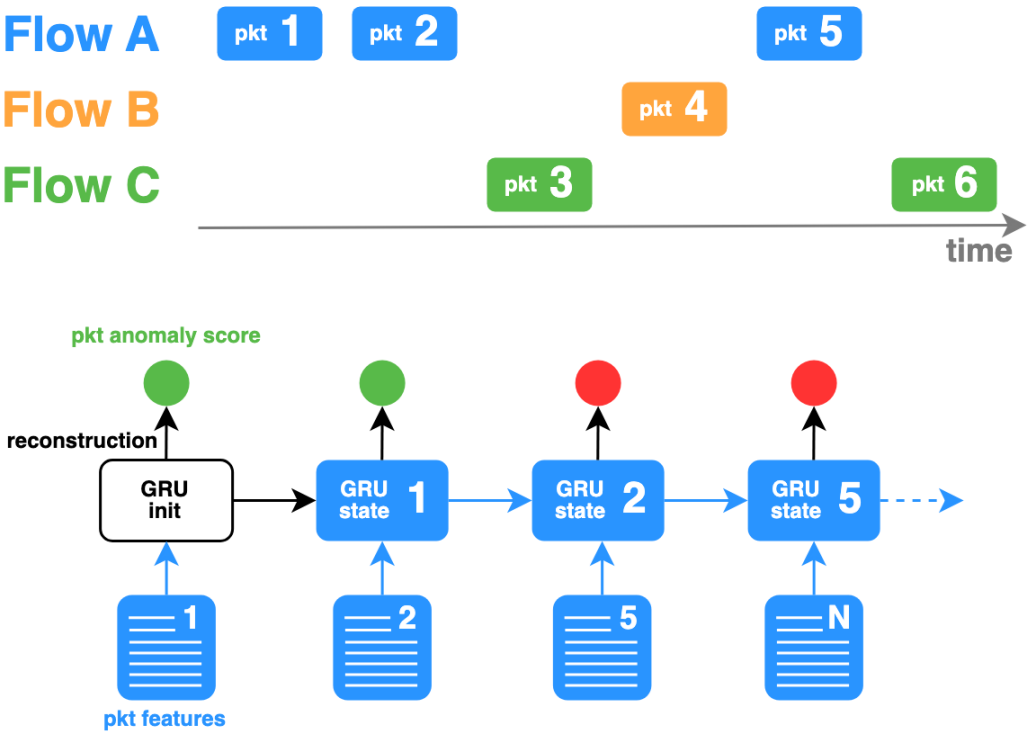

Figure 4.2: Proposed Gated Recurrent Autoencoder workflow

4.1.3 Model #3: Deep Gaussian Estimator

When compared with the previous models, this third model we investigate removes the computation of the decoder layer, and instead of reconstructing the input features it will individually analyze the resulting latent vector from each packet. On the one hand, this model is restricted to analyzing the latent space and may be less capable of abstracting complex network interactions, slightly losing on detection

performance. On the other, this model has a lighter representation, consuming fewer computational resources. The process is as follows:

1. Latent Vector Analysis

After the computation of the latent vector through the model encoder, which follows the same encoder formulation as in the Autoencoder (AE), the resulting latent vector will be mapped into a multivariate gaussian distribution $\mathcal{N}(0, I)$. The main objective of introducing this distribution is to analyze the latent vector through a well-structured distribution, that will define the anomaly probability based on the likelihood of each vector on this multivariate distribution. Therefore, similarly to AEs, the Deep Gaussian will still take advantage of learning non-linear correlations along the way.

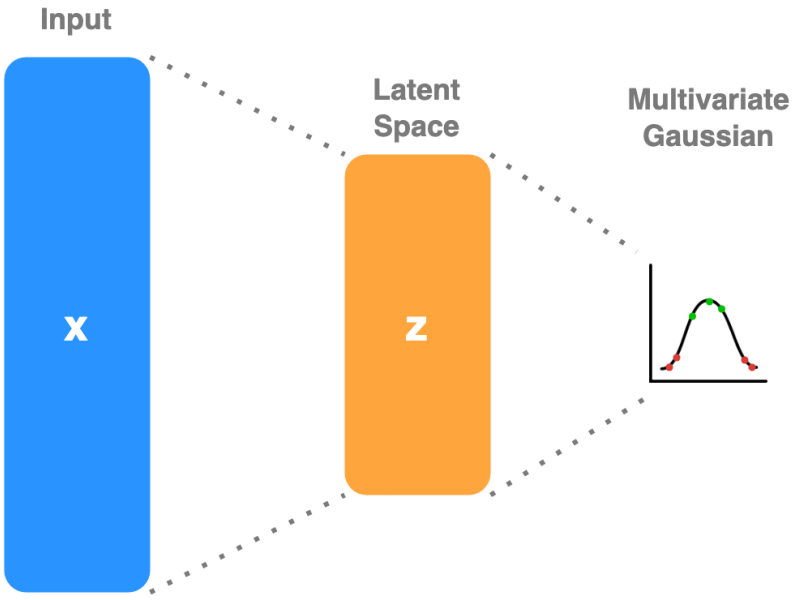

Figure 4.3: Proposed Deep Gaussian architecture

2. Training Phase

The fundamental assumption in our Deep Gaussian Estimator is that the data points are generated from a multivariate Gaussian distribution. The model will typically define as abnormal the data points that escape from the high density regions of the distribution, and consider as normal the points present in these specific areas. Having this, the objective during the training process will be to approximate the distribution of data points to a Gaussian-like shape. In our approach, this was achieved by using an approximation of the original Kullback-Leibler (KL) formulation (Equation 2.4) to measure the difference between the two distributions, and thus backpropagate the model to make these distributions alike in probabilistic terms. This way, incoming data about the network packets can be easily tested against a multivariate Gaussian, capturing anomalies that tend to deviate significantly from this distribution.

4.2 Feature Importance

Most feature extractors implement the calculation of a wide variety of feature statistics. In our case, we use Peregrine's feature extractor to feed the three ML models we selected to explore. However, depending on the profile of the attacks, some of the features may be more relevant than others. Most of the time, filtering those features allows the model to reduce its complexity and achieve even better detection results since irrelevant, sometimes noisy, features will be kept out from the training set. Consequently, these features will not disturb the anomaly detector, softening the learning process.

4.2.1 Calculated Features

In our thesis, we use Kitsune's feature extractor that implements various 1D and 2D statistics (Table 4.1), computing a total of 115 features. While 1D statistics keep track of the outbound traffic (i.e., i $\rightarrow$ j flow directions), 2D statistics deal with both outbound and inbound traffic (i.e., i $\rightarrow$ j and j $\rightarrow$ i flow directions). In order to calculate a panoply of features multiple flow keys are taken into account, namely [*src MAC*, *src IP*], [*src IP*], [*Channel*] and [*Socket*]. Additionally, in order to penalize and gradually forget the older occurrences, Kitsune introduces a decay function that is defined in Equation 4.2. The λ parameter in the equation represents the decay factor and regulates how fast older packets are forgotten, and t is the time that has passed since the last packet for the given flow key. To further add diversity, 5 different time windows are considered when extracting these features ($\lambda = 5, 3, 1, 0.1, 0.01$).

$$d_\lambda(t) = 2^{-\lambda t} \tag{4.2}$$

4.2.2 Model Dimensionality

In our approach, we tested both Random Forests (RFs) [Breiman, 2001] and XGBoost (XGB) [Chen and Guestrin, 2016] to search for the best selection of features. This selection process reduces the number of used features based on their importance score regarding the profile of the attack. Thus, it will reduce the dimensionality of the input layer leading to a more efficient execution of the model. However, the dimensionality of the input layer has a major impact on the learning phase. Selecting a very restricted number of features would lead to underfitting to training data and, in that case, the model might not be able to acquire a foundation knowledge about the normal patterns of the network. Therefore, the selection of important features is just another instance of the bias-variance trade-off. As a result, it is common in the literature to see this selection performed empirically.

Type	Statistic	Notation	Calculation
1D	Weight	w	w
	Mean	μ_{S_i}	LS/w
	Std. Deviation	σ_{S_i}	$\sqrt{\lvert SS/w - (LS/w)^2 \rvert}$
2D	Magnitude	$\lvert\lvert S_i, S_j \rvert\rvert$	$\sqrt{\mu_{S_i}^2 + \mu_{S_j}^2}$
	Radius	R_{S_i,S_j}	$\sqrt{(\sigma_{S_i}^2)^2 + (\sigma_{S_j}^2)^2}$
	Approx. Covariance	Cov_{S_i,S_j}	$\frac{SR_{ij}}{w_i + w_j}$
	Corr. Coefficient	CP_{S_i,S_j}	$\frac{Cov_{S_i,S_j}}{\sigma_{S_i}\sigma_{S_j}}$

LS = linear sum of the packet sizes
SS = squared sum of the packet sizes
SR = sum of residual products for streams i and j

Table 4.1: Computed features [Mirsky et al., 2018]

4.3 Implementation

The three lightweight models investigated in this thesis were implemented to be deployed on the control plane, where the model is going to be trained, using *PyTorch* [PyTorch Team, 2016], which can benefit from GPU support, and in the data plane with Taurus [Tang et al., 2016], where ML inference will take place. The Taurus logic can be customized using Spatial [Koeplinger et al., 2018], a Domain Specific Language (DSL) for accelerators, such as the FPGA Taurus used to prototype its architecture [Swamy et al., 2022a]. Spatial uses pipe blocks that allow the execution and customization of the *MapReduce* algorithm within Taurus. In this thesis, all implemented models will therefore recur to these blocks of computation. In the next subsections, we present one example in Spatial and another in *PyTorch*.

Consequently, we can separate the three aforementioned models into two categories: **Reconstruction-based Anomaly Detection** and **Latent Analysis-based Anomaly Detection**. Table 4.2 provides an overview of the three implemented models.

Model	Section	Category	Sequential Modeling
Autoencoder	4.1.1	Reconstruction-based	✗
Gated Recurrent Autoencoder	4.1.2	Reconstruction-based	✓
Deep Gaussian Estimator	4.1.3	Latent Analysis-based	✗

Table 4.2: Implemented models summary

4.3.1 Reconstruction-based Anomaly Detection

The first class of models implemented in this thesis are reconstruction-based models. Both the Autoencoders (AEs) and Gated Recurrent Autoencoder fall into this category. Given the example of the AE (Listing 4.1), we can separate reconstruction-based models into three pipelines (i.e. `Pipe` blocks in Spatial) that will respectively leverage the parallel computations of each model component (i.e., encoder, decoder and RMSE computation). Specifically, since DL models rely on repeated matrix multiplications, the operator `reduceTree` is crucial to perform reduction along the desired axes, efficiently leveraging SIMD parallelism.

Listing 4.1: Autoencoder (AE) implementation in Spatial [Koeplinger et al., 2018]

```
1  Pipe { // build latent space (ENCODER)
2      List.tabulate(L_DIM) { i =>
3          val partial_results = List.tabulate(I_DIM) { j =>
4              W(j, i) * (
5                  (input(MAP(j)) - NORM_MIN(j))
6                  /
7                  (NORM_MAX(j) - NORM_MIN(j) + 1e-15)
8              )
9          }
10         Z_OUT(i) = activation(partial_results.reduceTree {_+_} + B1(i))
11     }
12 } Pipe { // reconstruct latent vector (DECODER)
13     List.tabulate(I_DIM) { i =>
14         val partial_results = List.tabulate(L_DIM) { j =>
15             W(i, j) * Z_OUT(j)
16         }
17         R_OUT(i) = activation(partial_results.reduceTree {_+_} + B2(i))
18     }
19 } Pipe { // compute RMSE
20     sqrt(List.tabulate(I_DIM) { i =>
21         calc_se(
22             (input(MAP(i)) - NORM_MIN(i))
23             /
24             (NORM_MAX(i) - NORM_MIN(i) + 1e-15), R_OUT(i), 2
25         )
26     }.reduceTree {_+_} / I_DIM)
27 }
```

4.3.2 Latent Analysis-based Anomaly Detection

Latent analysis-based models, on the other hand, are restricted to analyzing and making conclusions entirely based on statistically testing the model's latent space. The Deep Gaussian Estimator was implemented to test the gains in terms of computation, by removing the typical decoder layer. Listing 4.2 presents part of the implementation of this lightweight model in *PyTorch*, which is used to train the model. As shown in line 4, similarly to Kitsune's implementation, we normalize all the input features so that they are between 0 and 1, facilitating the training process. These normalized features are then fed to the encoder network to compute the latent distribution vector (line 8). Finally, the KL divergence is calculated given the latent representation (line 10). However during evaluation instead of computing the KL divergence, we calculate the logarithmic density probability (line 28). The logarithmic probability turns out to be more efficient since it allows us to avoid computing the exponential.

Listing 4.2: Deep Gaussian Estimator implementation in *PyTorch*

```
1  def forward_train(self, x):
2    self.train()
3    # normalize input
4    x_norm = (x - self.norm_min) / (self.norm_max - self.norm_min + 1e-15)
5    # reset gradients
6    self.optimizer.zero_grad()
7    # run encoder
8    z = self.encoder(x_norm)
9    # KL Divergence as the loss
10   kl_div = self.kl_divergence(z)
11   # backprop
12   kl_div.backward()
13   self.optimizer.step()
14
15 def kl_divergence(self, z):
16   return -0.5 * torch.sum(
17     1 + torch.log(z.var(dim=0)) - z.mean(dim=0)**2 - z.var(dim=0)
18   )
19
20 def forward_eval(self, x):
21   self.eval()
22   # normalize input
23   x_norm = (x - self.mean_t) / (self.std_t + 1e-15)
```

```
    # run encoder
    with torch.no_grad():
      z = self.encoder(x_norm)
    # calculate likelihood
    return - self.gaussian.log_prob(z)
```

4.4 ML-based NIDS Workflow

We showcase the overall workflow of our system (Figure 4.4). We completely uncouple the programmable data plane from the slow control plane in our proposed NIDS, to be able to perform detection entirely in the network data plane, maintaining high-performance data forwarding. So, we can separate our workflow into two stages: **offline configuration stage** and **online runtime stage**. A more detailed explanation of these two stages of the workflow follows.

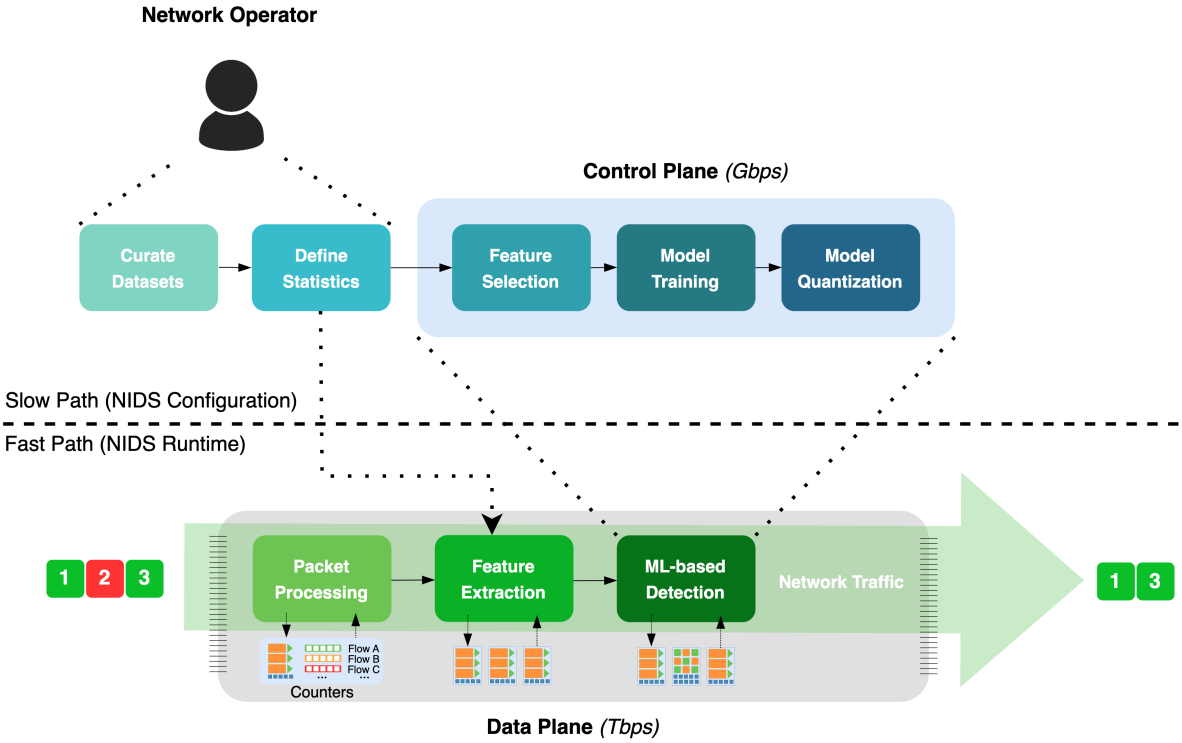

Figure 4.4: ML-based NIDS proposed workflow

4.4.1 Offline Configuration Stage

To support the deployment of the ML model to the network data plane, several steps take place. This process is done off the data plane since it requires hand-tuning phases, namely dataset curation and statistics definition. On top of that cumbersome executions, such as feature selection, model training and quantization will take place in the control plane, as each of these requires a significant amount of hardware resources.

1. Curate Datasets

In our thesis we use a set of datasets to train the models as well we select a generic set of statistics to be computed by the feature extractor. However, these modules are customizable, and at times is up to the network operator to define the datasets. In fact, ML models are data-hungry, and data curation ensures the cleaning and organization of data, which can help remove noise, errors, and inconsistencies, therefore reducing the risk of having inaccurate or misleading information in the dataset.

2. Statistics Definition

In addition to data curation, distinct statistics can be calculated depending on the profile of the attack or on the structure of the network. Consequently, the network operator can play a crucial role in crafting specific statistics to fit the NIDS' circumstances and needs. Crafting these statistics allows for a more customized approach to network security, enabling the NIDS to detect both known and emerging threats effectively. Moreover, network operators can leverage their domain expertise to identify patterns, anomalies, or specific indicators of compromise that may not be readily apparent through automated algorithms alone, but in fact, that could enrich the ML detection performance.

3. Feature Selection

Feature selection can be performed through feature importance analysis, and is the preliminary step in the ML model deployment (described in section 2.5.1). It involves identifying and selecting the most relevant features (input variables) for the classification task, which helps reduce the dimensionality of the data, improving efficiency and leading to faster inference times. Consequently, the dataset is pruned based on the selected set of features that will be passed to the next and core stage of deploying an ML model, the training phase.

4. Model Training

During this stage, the model learns patterns and relationships from normal network behavior on the training data to make predictions on new, unseen data. Model training is a critical phase, where the NIDS acquires the ability to recognize and differentiate between benign and malicious network activity. The accuracy and generalization of the model heavily depend on the previous steps that influence the

quality and diversity of the training dataset.

5. Model Quantization

The ML workflow can optionally be integrated with model compression techniques, such as quantization. Quantization helps optimize the model for efficient deployment, reducing memory and computational requirements, to ensure it can be deployed on the target hardware, by shrinking the model size. In this regard, we explored some quantization schemes using Post-Training Quantization (PTQ) and Quantization Aware Training (QAT). This allows a universal deployment with mainstream quantization methods. One important detail to point out is that both model training and quantization stages can be merged together, for instance, if one uses QAT. After this sequence of steps, the model is ready to be deployed to the network switch (e.g., Taurus [Swamy et al., 2022a]), on which anomaly detection will occur during runtime.

4.4.2 Online Runtime Stage

Later, after the offline configurations and optimizations in the control plane, we can successfully deploy the ML model to the network data plane. Since we run both feature computation and ML detection online, inside the data plane, the communications with the control plane are very limited, and they may only occur once the model is updated. This prevents the control plane from being a bottleneck in intrusion detection. The online runtime stage is therefore composed of the following steps.

1. Packet Processing

During the packet processing phase, incoming packets are identified by their flow key, a unique identifier used to distinguish and categorize network traffic into distinct flows. Flows in networking represent a sequence of packets between a source and destination with certain shared characteristics, such as the same source and destination IP addresses, source and destination port numbers, and the transport protocol (specifically, TCP or UDP). We use Peregrine [Amado et al., 2023] to keep a comprehensive set of flow counters (i.e., number of packets (w), number of bytes (LS), squared number of bytes (SS)) to allow the feature extractor to capture a wide variety of rich statistics to serve as input to our ML-based detection system.

2. Feature Extraction

Given the flow counters from the packet processing module, the feature computation consists of extracting and computing flow and packet-level statistics of varying levels of complexity, including packet length, header fields, and a variety of flow-based statistics (e.g., mean packet size, see Table 4.1) powered by Peregrine [Amado et al., 2023]. Rather than just relying on simple raw packet headers, learning-based detection systems can use these richer features to learn normal network behavior and detect devia-

tions from the norm more accurately for various types of network intrusions, anomalies, and security threats. However, most of these statistics require arithmetic operations that cannot be performed in the switch (e.g., multiplication, division). Therefore, some statistics often need to be approximated, affecting accuracy, a throughput/accuracy trade-off.

3. ML Detection

In this final phase, the ML model utilizes the extracted features to classify the network packets as benign or anomalous. This classification is performed in real-time as packets arrive at the network switch, allowing for swift identification of potential security threats and network intrusions.

4.5 Summary

In this chapter, we mainly discuss the models and techniques we propose to investigate in this thesis. We present three distinct DNN models, which are summarized in Table 4.2, and offer a brief overview of their implementation in both the control plane, which pertains to the training stage, and the Taurus data plane [Swamy et al., 2022a], where the models will be deployed. Additionally, we introduce the concept of feature importance as a valuable technique for further enhancing model performance. We conclude this chapter by elaborating on a ML-based workflow designed for NIDS.

Chapter 5

Experimental Results

The experiments in this chapter are driven by the need to enhance network security through the development of robust Network Intrusion Detection Systems (NIDS). The primary objective of this research is to construct a versatile system for anomaly detection in network traffic data using machine learning techniques. In order to assess the effectiveness and adaptability of our models in the network security context, we explore their performance on different datasets and optimization methods. Hence, this experimental evaluation section aims to answer the following research questions:

Q1: What is the hardware consumption of the proposed models on Taurus?

Q2: How does our approach compare against other NIDS?

Q3: Does a better selection of features improve detection performance?

Q4: How does model dimensionality affect performance?

Q5: What is the effect of quantization on system performance?

Q6: Are the existing available datasets enough to train a robust NIDS?

5.1 Implementation & Hardware

In this thesis, all the models were implemented and trained using the *PyTorch* deep learning library for Python [PyTorch Team, 2016]. The training was performed using a single Nvidia GeForce GTX 1060 graphics processing unit with 3GB of VRAM memory, in a machine with an AMD Ryzen 3 2200G processor and 16GB of DDR4 RAM. By simply implementing the Kitsune's [Mirsky et al., 2018] training script in *PyTorch* (i.e., providing GPU acceleration) we were able to improve the training speed by 2 orders of magnitude over the open-sourced Kitsune implementation. Subsequently, to execute our models in

Taurus [Swamy et al., 2022a], we also implement it using Spatial [Koeplinger et al., 2018] and compile the models with Vivado 2020.2 to make sure they run on the prototyped Taurus testbed (i.e., which uses an Xilinx Alveo U250 FPGA). To reproduce the results of the MapReduce block, we used a Taurus simulator, available in this repository [Swamy et al., 2022b]. In addition, the resulting and compiled FPGA designs were simulated on the Vivado IDE. We maintain the same Taurus configuration for the Compute Units (CUs), containing 16 SIMD lanes and 4 stages each, and therefore a CU in our approach will require roughly 0.044 mm^2 in area.

Datasets

Similarly to Peregrine [Amado et al., 2023], our system's evaluation is performed using attack traces sourced from two key datasets: (1) the Kitsune evaluation dataset [Mirsky et al., 2018] and (2) the CIC-IDS 2017 and 2018 datasets [Sharafaldin et al., 2018]. These datasets encompass meticulously annotated attacks within a real-world network laboratory environment and are part of a broader collection of datasets recognized for evaluating the efficacy of intrusion detection systems. The training stage of the ML Classifier module is always performed with benign traffic from the same dataset as the tested attack. For each trace, the initial packets represent benign traffic and serve as the training data for the model. The subsequent portion of the trace, which is composed of malicious traffic associated with a particular attack, is utilized to assess the detection performance of the trained classifier.

Metrics

The assessment of an algorithm's detection performance on a specific dataset can be quantified by its true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). Additionally, the system performance can also be gauged in terms of efficiency. So, in order to measure the detection performance of our models, we use the following metrics:

- **F-Score** is computed as the harmonic mean between the recall (true positive rate, which is the percentage of detected positives) and precision scores (percentage of accurate positives with regard to the detected positives). Higher f-scores mean there will be a small number of false alarms while maintaining high detection accuracy.
- **Recall score at FPR** is basically a metric to calculate the recall score ($TPR = \frac{TP}{TP+FN}$), fixing the False Positive Rate ($FPR = \frac{FP}{FP+TN}$). As a matter of fact, this value can be directly fetched from the ROC curve.
- **AUC-ROC Curve** is the area under the ROC curve, which typically shows the performance of the anomaly detector at all different thresholds settings (false positive rate values), given the anomaly probability of all data points. With this metric, a perfect classifier would split all the data points by its class perfectly, for at least one of the possible threshold settings, therefore achieving an AUC-ROC score of 1.

- **Chip Area** is measured as the necessary area to run the given model on the Taurus switch.
- **Power Consumption** is an estimation of the consumed power on Taurus for the given model.

5.2 Results & Analysis

Model	Precision	CUs	Area (mm^2)	Power (mW^2)
TAURUS AD	`fix8`	22	1.0	647
TAURUS LSTM	`fix8`	65	3.0	1897
KITSUNE	`fix8`	250	11.0	7296
Deep Gaussian	`fix8`	5	0.2	146
Autoencoder	`fix8`	13	0.6	379
Gated Recurrent Autoencoder	`fix8`	30	1.3	876
TAURUS AD	`fix32`	22	3.9	2568
TAURUS LSTM	`fix32`	65	11.4	7588
KITSUNE	`fix32`	250	44.0	29184
Deep Gaussian	`fix32`	5	0.9	584
Autoencoder	`fix32`	13	2.3	1518
Gated Recurrent Autoencoder	`fix32`	30	5.3	3502

Table 5.1: Hardware consumption (fixed-point precision `fix8` and `fix32`)

The chip area and power consumption are crucial considerations in Single Instruction Multiple Data (SIMD) architectures, such as the one we test in our thesis, Taurus [Swamy et al., 2022a]. SIMD architectures are designed to process multiple data elements in parallel, which often involves having multiple Compute Units (CUs) and widening data paths. Balancing the trade-off between performance gains and the associated chip area and power consumption is a fundamental challenge in the design and optimization of SIMD architectures. For that reason, Table 5.1 compares our proposed models with both Taurus prototyped models, an anomaly detection DNN and a LSTM. In addition, we calculate the hardware consumption of the Kitsune ensemble model, if it were to be deployed in Taurus. We assume our models have a latent space of size 2 (i.e., $dim(\mathbf{z}) = 2$) and we select the 10 features that best represent each attack. The next sections in this chapter will better support and explain this decision.

The chip area and power consumption calculation can be achieved by calculating the number of Compute Units (CUs) required by the given model. In Taurus, each CU has 16 SIMD lanes which execute the element-wise multiplication of a neuron. If a neuron in the DNN has no more than 16 weights, all the neuron computation can be performed within a CU. The fact a CU in Taurus has 4 stages is handy

since typically the computation of a neuron in DNNs can be separated into 4 distinct parts: (1) element-wise multiplication, (2) reduction (i.e., to sum the outputs of element-wise multiplication to form the inner product), (3) apply the bias, and (4) apply the activation function. However, in case a neuron depends on more than 16 weights, it will require an additional CU to execute all the multiplications and then one more CU to add all the results together. Having that said, after calculating the required number of CUs, the total area can be achieved by multiplying the number of CUs with the area of each CU for the 8-bit and 32-bit fixed-point scales, respectively 0.044 mm^2 and 0.176 mm^2. For the power the former consumes 29.184 mW^2, and the latter 116.736 mW^2.

The required area of Kitsune makes us conclude that it is practically unfeasible to run its ensemble model on Taurus. On the other hand, our models require a more reasonable amount of chip area when compared to the prototyped Taurus models. Compared with Taurus, our trained autoencoder and deep Gaussian estimator fall below the Anomaly Detector (AD) hardware consumption which can be executed at line-rate speed. In addition, our gated recurrent autoencoder requires less area and power when compared to the Taurus LSTM, gaining on the purpose of using RNNs.

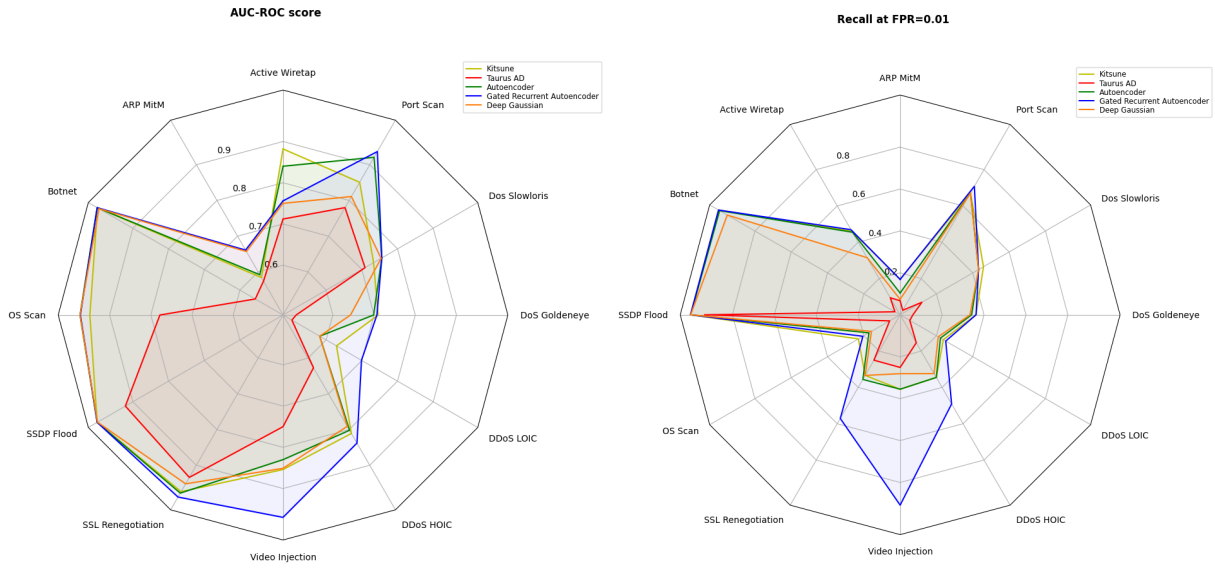

Figure 5.1: AUC-ROC score and Recall at FPR=0.01 comparison with Kitsune and the Taurus AD

The target plots in Figure 5.1 compare the Area Under the Curve (AUC) and recall scores results of our intrusion detector against both Kitsune [Mirsky et al., 2018] and Taurus [Swamy et al., 2022a]. One may conclude that our proposed models are able to perform neck and neck with Kitsune, presenting negligible performance degradation, and for some of the attacks still outperform it. Additionally, our approach clearly surpasses the prototyped Taurus anomaly detector. However, for some of the attacks,

namely *DDoS HOIC* and *ARP MitM*, all the models struggle to learn concrete patterns about the profile of the attack. We further discuss this problem in Section 5.6. To support our hypothesis that lighter models are still capable of learning rich representations regarding the profile of the attacks, we better visualize the latent space in Figures 5.2. The latent space is visualized by directly extracting the 2 latent variables from the latent space. However, as shown in Figure 5.3, we can apply Principal Component Analysis (PCA) to better represent the latent space. While the figures on the left represent the data the model was trained on (i.e., only containing benign instances), the figures on the right are both composed of anomalous and normal packets (i.e., test set). For both *Botnet* and *SSDP Flood* attacks, a clear pattern between benign and anomalous packets is formed, being an evident sign that lightweight models in NIDS are able to create a clear distinction between packets.

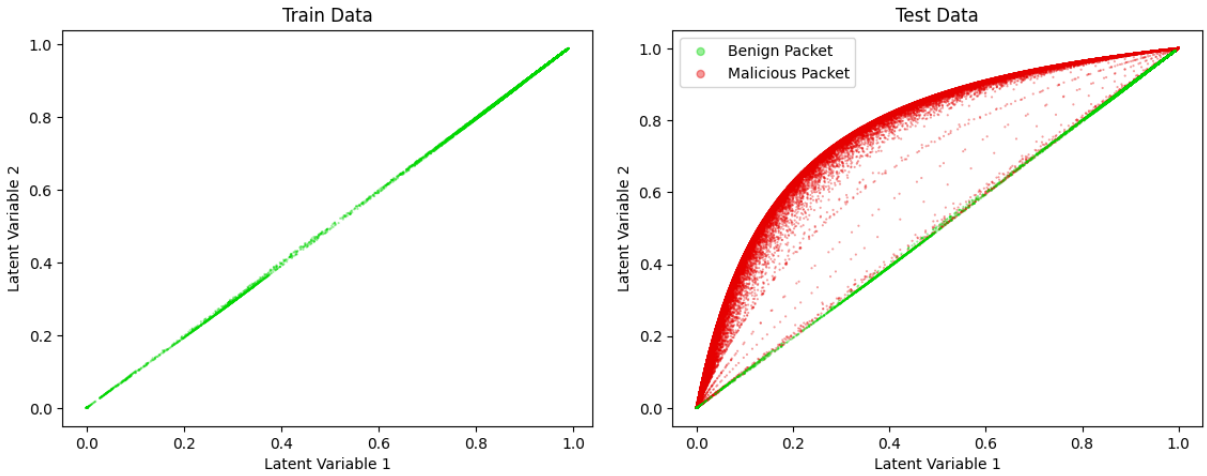

Figure 5.2: Latent-space visualization for the *Botnet* attack

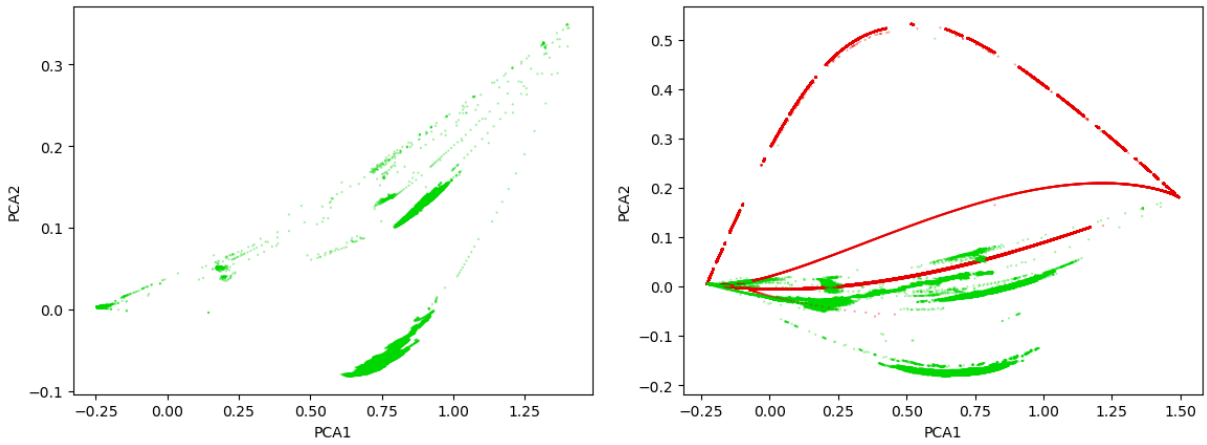

Figure 5.3: Latent-space visualization for the *SSDP Flood* attack with PCA

5.3 Feature Importance

Understanding the relevance of individual features is a critical aspect, for it has a direct bearing on the model's capability to render precise classifications. Indeed, ML models can sometimes grapple with irrelevant features, and this susceptibility is often due to their initialization. An unfortunate initialization can lead the model to fixate on specific, often irrelevant features, and become *stranded* in a local minimum or maximum along the optimization, leading to a suboptimal model. From our empirical evaluation, such fixation can reduce the model's capacity to effectively filter out noisy features and extract relevant information. In addition, reducing the number of features based on their importance results in a smaller feature space. Thus, this process allows for the selection of a more compact and informative set of input features, as well as reducing the model complexity.

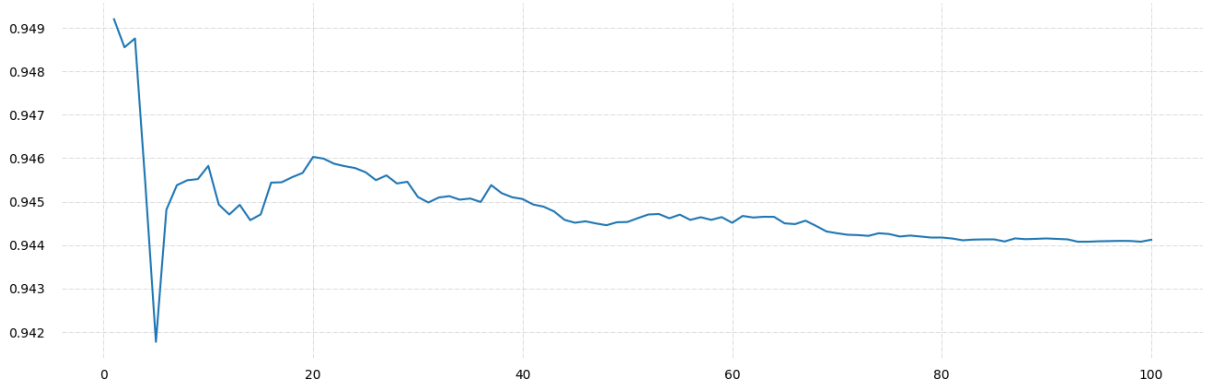

Figure 5.4: Performance degradation on *Port Scan* attack (from the best to the worst features)

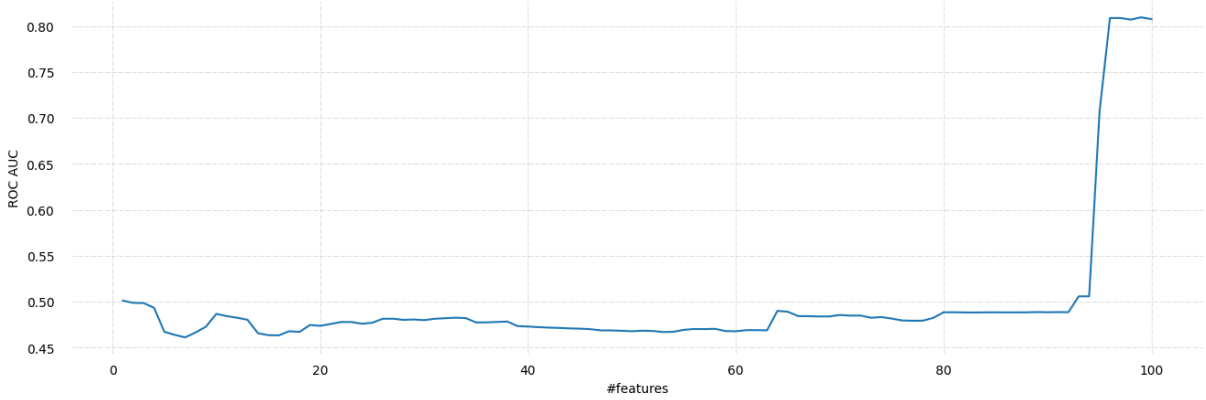

Figure 5.5: Performance improvement on *DDoS HOIC* attack (from the worst to the best features)

To support this conclusion, we present two charts (Figures 5.4 and 5.5) that showcase: (1) the degradation of the model's detection performance when incrementing the number of best features to select for the *Port Scan* attack, and (2) the evolution in detection performance when going from the

worst features to the best ones for the *DDoS HOIC* attack. In fact, by observing the second chart it is actually discernible that training a model for the *ARP MitM* attack based on the *less important* statistics, even with a large subset of features, will only lead to a detection performance around 0.5 of AUC, which will make the model perform as accurate as a random classifier. In addition, appendix A better explains the feature relevance across all the attacks.

As a result, Table 5.2 demonstrates the reflected improvements in AUC score when selecting specific amounts of features, and for most of the attacks, we were able to avoid detection performance degradation. Thus instead, we not only improved the detection performance of the ML intrusion detection system, but the main point is that we actually were able to significantly compress the model itself leading to an overall efficient execution. So, based on these results and the hardware limitations, we believe selecting the most relevant 10 features for each attack has the best trade-off between detection performance and efficiency.

Model	N Best Features	Avg. AUC Improv. (%)	Compression (%)
Autoencoder	115	0.00 %	0.00 %
Autoencoder	25	2.08 %	77.99 %
Autoencoder	10	4.05 %	90.99 %
Deep Gaussian	115	0.00 %	0.00 %
Deep Gaussian	25	2.92 %	77.59 %
Deep Gaussian	10	5.98 %	90.52 %
Gated Recurrent Autoencoder	115	0.00 %	0.00 %
Gated Recurrent Autoencoder	25	8.87 %	76.34 %
Gated Recurrent Autoencoder	10	17.60 %	89.07 %

Table 5.2: Average relative improvement across the different attacks when selecting the *N* best features

5.4 Model Dimensionality

When encoding features to a lower-dimensional latent space, it is important to address the impact of dimensionality reduction. Studying the relevance importance is definitely a step in the right direction that will eventually lead to a significant reduction of complexity. On the other hand, the main point of the ML model latent space is to have a light and intermediate representation of the input features. With this regard, in intrusion detection, most of the computed features are not fully independent and somehow correlate with each other (e.g., through the decay factor). This makes us hypothesize that a smaller latent space will still be able to accurately encode the input vector. In Kitsune, [Mirsky et al., 2018] the size of this intermediate layer of the neural network is still set to 75% of the input space, which we actually

believe is unreasonable. Indeed, our findings are that going further and reducing the dimensionality of the latent space does not incur major detection losses, but instead, it is able to compress and reduce the complexity of the model (Table 5.3). In general, for the tested datasets, reducing to as low as a 2-dimensional latent vector is enough to sustain a reasonable detection degradation, while compressing the model up to roughly 80% of the initial model size (i.e., with regard to the number of model parameters). Therefore, the dimension of the latent space appears as an important parameter that can be translated into a more efficient ML execution in the Taurus architecture.

Model	dim(z)	Avg. AUC Degradation (%)	Compression (%)
Autoencoder	8	0.00 %	0.00 %
Autoencoder	5	0.47 %	27.97 %
Autoencoder	2	0.77 %	55.93 %
Deep Gaussian	8	0.00 %	0.00 %
Deep Gaussian	5	1.99 %	37.50 %
Deep Gaussian	2	2.49 %	75.00 %
Gated Recurrent Autoencoder	8	0.00 %	0.00 %
Gated Recurrent Autoencoder	5	0.16 %	44.64 %
Gated Recurrent Autoencoder	2	0.29 %	79.93 %

Table 5.3: Average relative degradation across the different attacks for different latent space dimensions dim(z)

5.5 Quantization

On top of reducing model dimensionality, the incorporation of quantization methods can hold a significant value within NIDS. Since the input space is not fully independent, this will make the model rely on weight matrices with low *intrinsic rank*. This opens a path to further compress the ML model through quantization. The main assumption is that the models will be able to endure a bit-wise compression that will bind their computations while keeping a solid knowledge about the domain of the attack. Table 5.4 presents the suffered AUC degradation for the Post-Training Quantization (PTQ) and Quantization Aware Training (QAT) methods. Additionally, we show the gains in terms of area and power, if it were to be deployed in Taurus. As a consequence, it will result in a significant inference speedup.

We implemented QAT with *PyTorch* [PyTorch Team, 2016] and PTQ with TVM [Chen et al., 2018], an open-source Deep Learning Compiler Stack for ML accelerators. For QAT we only tested 8-bit quantization due to the lack of support for the lower bit scales. However, naturally QAT was able to better withstand compression due to the fact training is performed with the quantized model. In contrast, PTQ

quantizes the model after its training and is exposed to a slightly larger degradation. Nevertheless, both approaches are able to bear quantization with minimal losses.

Method	Quantization	Avg. AUC Degradation (%)	Area & Power
PTQ	8-bit	0.65 %	– 75.0 %
PTQ	4-bit	0.82 %	– 87.5 %
PTQ	3-bit	1.23 %	– 90.6 %
QAT	8-bit	0.50 %	– 75.0 %

Table 5.4: Relative degradation across the different attacks with quantization, compared to full precision (32-bit)

5.6 Discussion

The main goal of this evaluation was to show that it is possible to perform as accurately as Kitsune [Mirsky et al., 2018] while consuming significantly fewer resources. We empirically presented methods that are able to either improve the overall detection performance of the anomaly detector, or in parallel increase the efficiency of the model execution. At the same time, we introduce a comprehensive estimation of the needed resources to run such models in the Taurus architecture [Swamy et al., 2022a] in order to look further and run ML-based Anomaly Detection at line-rate.

In the aftermath of this chapter, it is important to note that some of the studied attacks are still difficult to extract insightful information from (e.g., CICIDS dataset). As a result, this unpractical outcome can prevent the general deployment of such NIDS. In fact, there is an important need to have a mainstream standardized dataset in the field of NIDS, like in many subjects of machine learning [Apruzzese et al., 2023]. At the same time, there is room to improve the set of features being computed by the current state-of-the-art NIDS. It seems strictly impossible for some of these attacks to simply rely on similar features that represent similar aspects within the network. Therefore, a broader scope of features may be necessary in order to leverage a quality representation regarding the profile of the attacks. As a consequence, scaling the models still seems to be compromised, and smaller models are still able to keep up with larger models, whereas in many other ML applications scalability has a direct impact on the performance of the model.

Chapter 6

Conclusion

In a new era marked by the rapid proliferation of cutting-edge networks and technologies, ML-based Network Intrusion Detection Systems (NIDS) appear as a prosperous field of research. These advanced systems are gaining prominence due to their ability to fortify network security in the face of increasingly sophisticated threats. ML-based NIDS leverage network data to enhance threat detection and response, thereby providing robustness to modern infrastructures. Their role in identifying and mitigating potential security breaches underscores their importance in safeguarding sensitive information and sometimes ensuring the uninterrupted operation of critical network systems. As our reliance on networks and technology deepens, the continued development and application of ML-based NIDS stand as a vital measure in maintaining the integrity and security of our digital landscape.

6.1 Final Remarks

In this thesis, we explored, implemented and evaluated different Deep Neural Network (DNN) models in the context of NIDS. Our primary emphasis was gaining a profound understanding of these models' capabilities and their potential to enhance the efficiency and effectiveness of NIDS. At the same time, our focus was to understand the possibility of deploying such models in the network data plane at line-rate speeds. In order to fulfill this requirement, we investigated a new PISA-based switch architecture, Taurus [Swamy et al., 2022a], that leverages SIMD parallelism to enable per-packet ML inference in the data plane. As a result, we were to conclude that it is possible to decrease the size of the model, significantly consuming fewer resources in Taurus, while getting leveled results when compared to Kitsune's approach [Mirsky et al., 2018], a state-of-the-art NIDS. In parallel, the models we explored outperform the generic Taurus Anomaly Detector (AD). We also investigated methods to run ML model on resource-constrained devices that have been successful in other fields of ML, such as feature selection and quantization.

We hope that this thesis will ignite insightful discussions on the design of enhanced switch architectures together with model compression techniques capable of efficiently running robust ML models without compromising their ability to operate at line-rate.

6.2 Limitations & Future Work

New Architectures

Although it is a step in the right direction, Taurus [Swamy et al., 2022a] is still restricted either to very small models or heavy compression techniques. For larger models, Taurus is not able to keep up with line-rate speeds, forcing it to resort to network sampling. We look forward to seeing if a new wave of hardware specialization in the paradigm of networking brings a diverse set of acceleration primitives for ML inference in the data plane.

New Datasets

Unlike most fields in ML, the shortage of widely accepted datasets in NIDS has been a persistent challenge in the field. This scarcity of comprehensive datasets hinders the development and evaluation of NIDS algorithms and models, making it difficult to gauge their real-world performance effectively. Besides, according to SoK [Apruzzese et al., 2023], some of the most used datasets in NIDS are even flawed. This creates the need for the establishment of standardized datasets in NIDS to ensure the reliability and efficacy of network security solutions.

Synthetic Data

In ML, synthetic data refers to artificially generated data that mimics real-world data but is created by statistical algorithms or simulations rather than collected from actual observations. It has been a crucial topic in various fields, such as healthcare [Chen et al., 2021], in mitigating issues related to data availability and privacy concerns, as it enables researchers to explore extensive datasets without compromising sensitive information. In the context of networking, NetShare [Yin et al., 2022] recently proposed a trace generator based on Generative Adversarial Networks (GANs). With the lack of available mainstream datasets in NIDS, this topic certainly emerges as an opportunity.

Compression Techniques

In addition to the compression techniques we explored in this thesis (e.g., quantization), other compression techniques such as pruning and distillation can optimize the efficiency and deployment of DNNs, and are worth being investigated in NIDS. Pruning involves reducing the network's size by selectively removing less influential neurons or connections, effectively streamlining the model while maintaining its performance. This technique significantly reduces the computational and memory requirements, bene-

fiting resource-constrained environments. On the other hand, distillation focuses on knowledge transfer, where a larger, well-trained model (teacher) transmits its knowledge to a smaller model (student). This process allows the student model to capture the essence of the teacher's foundation knowledge, achieving compression without substantial loss in accuracy.

Bibliography

[Alcoz et al., 2022] Alcoz, A. G., Strohmeier, M., Lenders, V., and Vanbever, L. (2022). Aggregate-based congestion control for pulse-wave ddos defense. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 693–706.

[Amado et al., 2023] Amado, J., Pereira, F. C., Signorello, S., Correia, M., and Ramos, F. (2023). Poster: In-Network ML Feature Computation for Malicious Traffic Detection. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 1105–1107.

[Apruzzese et al., 2023] Apruzzese, G., Laskov, P., and Schneider, J. (2023). Sok: Pragmatic assessment of machine learning for network intrusion detection. *arXiv preprint arXiv:2305.00550*.

[Arp et al., 2022] Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. (2022). Dos and don'ts of machine learning in computer security. In *Proc. of the USENIX Security Symposium*.

[Atre et al., 2022] Atre, N., Sadok, H., Chiang, E., Wang, W., and Sherry, J. (2022). Surgeprotector: Mitigating temporal algorithmic complexity attacks using adversarial scheduling. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 723–738.

[Barradas et al., 2021] Barradas, D., Santos, N., Rodrigues, L., Signorello, S., Ramos, F. M., and Madeira, A. (2021). Flowlens: Enabling efficient flow classification for ml-based network security applications. In *NDSS*.

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

[Boutaba et al., 2018] Boutaba, R., Salahuddin, M. A., Limam, N., Ayoubi, S., Shahriar, N., Estrada-Solano, F., and Caicedo, O. M. (2018). A comprehensive survey on machine learning for networking: evolution, applications and research opportunities. *Journal of Internet Services and Applications*, 9(1):1–99.

[Branco et al., 2020] Branco, B., Abreu, P., Gomes, A. S., Almeida, M. S., Ascensão, J. T., and Bizarro, P. (2020). Interleaved sequence rnns for fraud detection. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 3101–3109.

[Breiman, 2001] Breiman, L. (2001). Random forests. *Machine learning*, 45:5–32.

[Cao et al., 2017] Cao, Z., Long, M., Wang, J., and Yu, P. S. (2017). Hashnet: Deep learning to hash by continuation. In *Proceedings of the IEEE international conference on computer vision*, pages 5608–5617.

[Chandola et al., 2009] Chandola, V., Banerjee, A., and Kumar, V. (2009). Anomaly detection: A survey. *ACM computing surveys (CSUR)*, 41(3):1–58.

[Chen et al., 2021] Chen, R. J., Lu, M. Y., Chen, T. Y., Williamson, D. F., and Mahmood, F. (2021). Synthetic data in machine learning for medicine and healthcare. *Nature Biomedical Engineering*, 5(6):493–497.

[Chen and Guestrin, 2016] Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794.

[Chen et al., 2018] Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. (2018). Tvm: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 579–594, USA. USENIX Association.

[Chung et al., 2014] Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*.

[Coralie et al., 2019] Coralie, Meier, R., Dietmüller, A., Bühler, T., and Vanbever, L. (2019). pforest: In-network inference with random forests. *arXiv preprint arXiv:1909.05680*.

[Fernando et al., 2021] Fernando, T., Gammulle, H., Denman, S., Sridharan, S., and Fookes, C. (2021). Deep learning for medical anomaly detection – a survey. *ACM Comput. Surv.*, 54(7).

[Firestone et al., 2018] Firestone, D., Putnam, A., Mundkur, S., Chiou, D., Dabagh, A., Andrewartha, M., Angepat, H., Bhanu, V., Caulfield, A., Chung, E., et al. (2018). Azure accelerated networking:{SmartNICs} in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66.

[Frantar et al., 2023] Frantar, E., Ashkboos, S., Hoefler, T., and Alistarh, D. (2023). Gptq: Accurate post-training quantization for generative pre-trained transformers.

[Fu et al., 2021] Fu, C., Li, Q., Shen, M., and Xu, K. (2021). Realtime robust malicious traffic detection via frequency domain analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3431–3446.

[Gholami et al., 2021] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W., and Keutzer, K. (2021). A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630.

[Hamza et al., 2019] Hamza, A., Gharakheili, H. H., Benson, T. A., and Sivaraman, V. (2019). Detecting volumetric attacks on lot devices via sdn-based monitoring of mud activity. In *Proceedings of the 2019 ACM Symposium on SDN Research*, pages 36–48.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8):1735–1780.

[Intel, 2016] Intel (2016). Intel® Tofino™ series programmable ethernet switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

[Koeplinger et al., 2018] Koeplinger, D., Feldman, M., Prabhakar, R., Zhang, Y., Hadjis, S., Fiszel, R., Zhao, T., Nardi, L., Pedram, A., Kozyrakis, C., et al. (2018). Spatial: A language and compiler for application accelerators. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 296–311.

[Kreutz et al., 2014] Kreutz, D., Ramos, F. M., Verissimo, P. E., Rothenberg, C. E., Azodolmolky, S., and Uhlig, S. (2014). Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76.

[Liu et al., 2021] Liu, Z., Namkung, H., Nikolaidis, G., Lee, J., Kim, C., Jin, X., Braverman, V., Yu, M., and Sekar, V. (2021). Jaqen: A {High-Performance}{Switch-Native} approach for detecting and mitigating volumetric {DDoS} attacks with programmable switches. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3829–3846.

[Mahadevan et al., 2010] Mahadevan, V., Li, W., Bhalodia, V., and Vasconcelos, N. (2010). Anomaly detection in crowded scenes. In *2010 IEEE computer society conference on computer vision and pattern recognition*, pages 1975–1981. IEEE.

[McKeown et al., 2008] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. (2008). Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74.

[Medhi and Ramasamy, 2017] Medhi, D. and Ramasamy, K. (2017). *Network routing: algorithms, protocols, and architectures*. Morgan kaufmann.

[Mirsky et al., 2018] Mirsky, Y., Doitshman, T., Elovici, Y., and Shabtai, A. (2018). Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*.

[Nagel et al., 2021] Nagel, M., Fournarakis, M., Amjad, R. A., Bondarenko, Y., van Baalen, M., and Blankevoort, T. (2021). A white paper on neural network quantization. *CoRR*, abs/2106.08295.

[Paxson, 1999] Paxson, V. (1999). Bro: a system for detecting network intruders in real-time. *Computer networks*, 31(23-24):2435–2463.

[Prabhakar et al., 2017] Prabhakar, R., Zhang, Y., Koeplinger, D., Feldman, M., Zhao, T., Hadjis, S., Pedram, A., Kozyrakis, C., and Olukotun, K. (2017). Plasticine: A reconfigurable architecture for parallel patterns. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 389–402. IEEE.

[PyTorch Team, 2016] PyTorch Team (2016). PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. https://pytorch.org.

[Quatrini et al., 2020] Quatrini, E., Costantino, F., Di Gravio, G., and Patriarca, R. (2020). Machine learning for anomaly detection and process phase classification to improve safety and maintenance activities. *Journal of Manufacturing Systems*, 56:117–132.

[Reynolds et al., 2009] Reynolds, D. A. et al. (2009). Gaussian mixture models. *Encyclopedia of biometrics*, 741(659-663).

[Roesch et al., 1999] Roesch, M. et al. (1999). Snort: Lightweight intrusion detection for networks. In *Lisa*, volume 99, pages 229–238.

[Sanvito et al., 2018] Sanvito, D., Siracusano, G., and Bifulco, R. (2018). Can the network be the ai accelerator? In *Proceedings of the 2018 Morning Workshop on In-Network Computing*, pages 20–25.

[Sapio et al., 2019] Sapio, A., Canini, M., Ho, C.-Y., Nelson, J., Kalnis, P., Kim, C., Krishnamurthy, A., Moshref, M., Ports, D. R., and Richtárik, P. (2019). Scaling distributed machine learning with in-network aggregation. *arXiv preprint arXiv:1903.06701*.

[Sharafaldin et al., 2018] Sharafaldin, I., Lashkari, A. H., and Ghorbani, A. A. (2018). Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp*, 1:108–116.

[Siracusano and Bifulco, 2018] Siracusano, G. and Bifulco, R. (2018). In-network neural networks. *arXiv preprint arXiv:1801.05731*.

[Sohi et al., 2021] Sohi, S. M., Seifert, J.-P., and Ganji, F. (2021). Rnnids: Enhancing network intrusion detection systems through deep learning. *Computers & Security*, 102:102151.

[Swamy et al., 2022a] Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., and Olukotun, K. (2022a). Taurus: A data plane architecture for per-packet ml. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 1099–1114, New York, NY, USA. Association for Computing Machinery.

[Swamy et al., 2022b] Swamy, T., Rucker, A., Shahbaz, M., Gaur, I., and Olukotun, K. (2022b). Taurus: MapReduce implementation in FPGA. https://gitlab.com/dataplane-ai/taurus/mapreduce.

[Tang et al., 2016] Tang, T. A., Mhamdi, L., McLernon, D., Zaidi, S. A. R., and Ghogho, M. (2016). Deep learning approach for network intrusion detection in software defined networking. In *2016 international conference on wireless networks and mobile communications (WINCOM)*, pages 258–263. IEEE.

[Tee and Taylor, 2020] Tee, J. and Taylor, D. P. (2020). Is information in the brain represented in continuous or discrete form? *IEEE Transactions on Molecular, Biological and Multi-Scale Communications*, 6(3):199–209.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Viroli and McLachlan, 2019] Viroli, C. and McLachlan, G. J. (2019). Deep gaussian mixture models. *Statistics and Computing*, 29:43–51.

[Wu et al., 2020] Wu, H., Judd, P., Zhang, X., Isaev, M., and Micikevicius, P. (2020). Integer quantization for deep learning inference: Principles and empirical evaluation. *arXiv preprint arXiv:2004.09602*.

[Xie et al., 2022] Xie, G., Li, Q., Dong, Y., Duan, G., Jiang, Y., and Duan, J. (2022). Mousika: Enable general in-network intelligence in programmable switches by knowledge distillation. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications*, pages 1938–1947. IEEE.

[Xu et al., 2018] Xu, H., Chen, W., Zhao, N., Li, Z., Bu, J., Li, Z., Liu, Y., Zhao, Y., Pei, D., Feng, Y., et al. (2018). Unsupervised anomaly detection via variational auto-encoder for seasonal kpis in web applications. In *Proceedings of the 2018 world wide web conference*, pages 187–196.

[Xu et al., 2021] Xu, J., Wu, H., Wang, J., and Long, M. (2021). Anomaly transformer: Time series anomaly detection with association discrepancy. *arXiv preprint arXiv:2110.02642*.

[Yang et al., 2019] Yang, J., Shen, X., Xing, J., Tian, X., Li, H., Deng, B., Huang, J., and Hua, X.-s. (2019). Quantization networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7308–7316.

[Yin et al., 2022] Yin, Y., Lin, Z., Jin, M., Fanti, G., and Sekar, V. (2022). Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 458–472, New York, NY, USA. Association for Computing Machinery.

[Zhang et al., 2020] Zhang, M., Li, G., Wang, S., Liu, C., Chen, A., Hu, H., Gu, G., Li, Q., Xu, M., and Wu, J. (2020). Poseidon: Mitigating volumetric ddos attacks with programmable switches. In *the 27th Network and Distributed System Security Symposium (NDSS 2020)*.

[Zhao et al., 2020] Zhao, Z., Sadok, H., Atre, N., Hoe, J. C., Sekar, V., and Sherry, J. (2020). Achieving 100gbps intrusion prevention on a single server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1083–1100.

[Zheng et al., 2022a] Zheng, C., Xiong, Z., Bui, T. T., Kaupmees, S., Bensoussane, R., Bernabeu, A., Vargaftik, S., Ben-Itzhak, Y., and Zilberman, N. (2022a). Iisy: Practical in-network classification. *arXiv preprint arXiv:2205.08243*.

[Zheng et al., 2022b] Zheng, C., Zang, M., Hong, X., Bensoussane, R., Vargaftik, S., Ben-Itzhak, Y., and Zilberman, N. (2022b). Automating in-network machine learning. *arXiv preprint arXiv:2205.08824*.

[Zhou et al., 2023] Zhou, G., Liu, Z., Fu, C., Li, Q., and Xu, K. (2023). An efficient design of intelligent network data plane. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6203–6220, Anaheim, CA. USENIX Association.

[Zuev and Moore, 2005] Zuev, D. and Moore, A. W. (2005). Traffic classification using a statistical approach. In *International workshop on passive and active network measurement*, pages 321–324. Springer.

Appendix A

Overall Feature Importance

Figure A.1 contains a bar chart with the relative importance of every computed feature averaged across the tested attacks. The importance was calculated using XGBoost (XGB) [Chen and Guestrin, 2016] that will basically decide what the gain of information of a given feature is, based on the prior predictions of the tree-based model. Typically the most relevant features include 2D calculations (i.e., flow-based features: `HH` and `HpHp`) with a small decay value which will force feature calculations over a broader time window. Each feature name in the figure follows this notation: {`flow key`}_{`decay factor`}_{`stat`}. We study feature importance for the following flow keys:

- `MI` (1D): MAC.IP / —
- `HH_jit` (1D): srcIP.dstIP / —
- `HH` (2D): srcIP / dstIP
- `HpHp` (2D): srcIP.proto / dstIP.proto

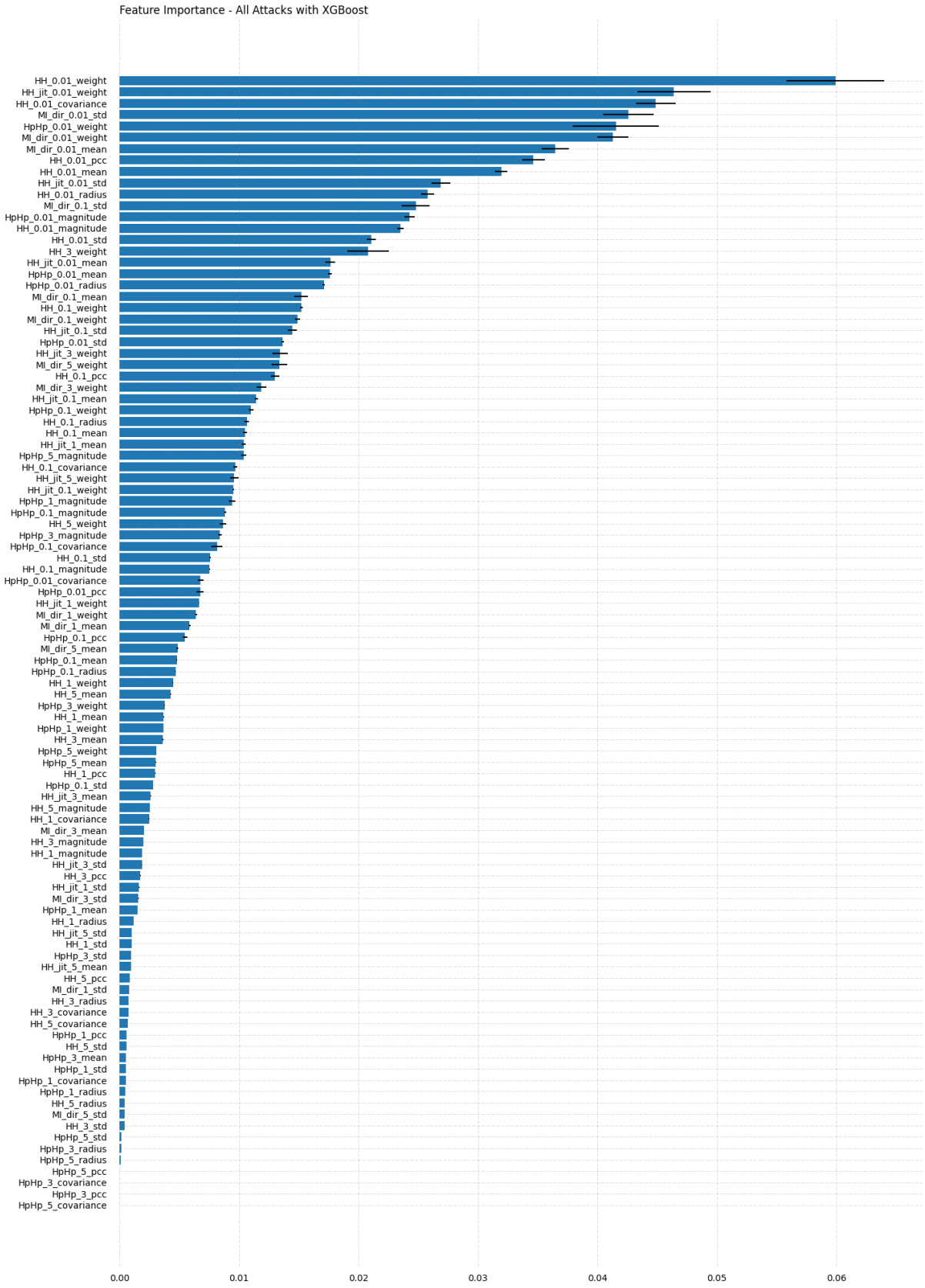

Figure A.1: Overall Feature Importance with XGBoost (XGB) [Chen and Guestrin, 2016]