Politecnico di Torino

Master's degree course in Cybersecurity

Master Degree Thesis

Learning to Defend: Adaptive Mitigation of Amplification-Based DDoS Attacks in Programmable Networks

Supervisors
Prof. Riccardo Sisto
Prof. Fulvio Valenza
Prof. Paola Grosso (UvA)
Dr. Marios Avgeris (UvA)
Dr. Anestis Dalgkitsis (UvA)

Candidate
Simone Sampognaro

Academic Year 2025-2026

Summary

Distributed Denial of Service attacks have evolved into persistent, large-scale threats to Internet availability, with amplification-based techniques enabling adversaries to generate terabit-per-second traffic volumes by exploiting protocol asymmetries. Traditional mitigation strategies, often relying on static filtering rules or control-plane-based inspection, struggle to cope with increasingly dynamic attack patterns. As a result, recent research has shifted toward In-Network Machine Learning, leveraging programmable data planes enabled by the P4 language to detect and mitigate attacks directly within network switches at line rate. While programmable switches allow customized packet parsing, telemetry extraction, and stateful processing inside the forwarding pipeline, strict line-rate constraints, limited memory, and restricted arithmetic capabilities significantly constrain the class of machine learning algorithms that can be implemented.

A review of existing in-network machine learning approaches shows that most prior work focuses on inference, while adaptive and online policy updates entirely within the programmable data plane remain largely unexplored. Motivated by this gap, we analyze candidate learning paradigms under these architectural constraints and identify reinforcement learning as a promising direction due to its incremental update capability and ability to operate with limited computational primitives.

Building on this insight, we investigate the feasibility of performing amplification-aware DDoS mitigation directly inside the programmable data plane through online reinforcement learning. We design a learning agent embedded within the packet processing pipeline that updates its mitigation policy without relying on external control-plane retraining or offline datasets. The agent operates on line-rate features derived from bidirectional flow telemetry, focusing on traffic asymmetry indicators between inbound and outbound flows and on a Quality-of-Service degradation metric computed at the switch.

The system is implemented as a P4-based prototype and evaluated within an emulated network environment using programmable software switches. Synthetic traffic is generated to emulate amplification attack scenarios and legitimate flows, enabling controlled experimentation of the in-switch learning mechanism. Experimental results show that the proposed approach can adapt to evolving traffic patterns and effectively reduce malicious amplification traffic while preserving legitimate service performance, highlighting the potential of programmable data planes for autonomous and scalable DDoS mitigation within network infrastructure.

Acknowledgements

I would first like to express my sincere gratitude to my technical supervisors, Dr. Marios Avgeris and Dr. Anestis Dalgkitsis, for their guidance throughout this work. Their constant availability, insightful advice, and support were invaluable for the development of this project. Over the course of this long and demanding journey, they became a true point of reference for me, and I deeply admire them as researchers.

I would also like to thank Professor Paola Grosso for welcoming me into the Multiscale Networked Systems research group and for making me feel part of it from the very beginning. The time spent working in such a stimulating environment is something I will always remember with great appreciation, and it contributed significantly to my growth both academically and personally.

My sincere thanks also go to my supervisors, Professors Riccardo Sisto and Fulvio Valenza, for their trust in giving me this opportunity.

Finally, I would like to express my deepest gratitude to my loved ones. My family has always been a constant source of support and encouragement throughout my academic path, providing the foundation that allowed me to reach this milestone. I am equally grateful to those who have shared this journey with me and have been present during both the challenging and the rewarding moments of these years.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context and Objectives

Distributed Denial of Service (DDoS) attacks continue to represent one of the most severe threats to Internet security and stability. Their core objective is to overwhelm a target's resources, whether computing power, memory, or bandwidth-until legitimate users can no longer access services. The open nature of the Internet makes such attacks particularly difficult to prevent: attackers can generate or redirect malicious traffic from distributed sources with relative ease, exploiting available infrastructure and tools to disrupt even well-protected organizations.

The scale and intensity of DDoS attacks have grown significantly in recent years. In the first quarter of 2025, Cloudflare reported blocking 20.5 million attacks, a 358% increase compared to the same period in 2024 [3]. By Q2 2025, this figure had risen to 27.8 million attacks, already surpassing the total volume observed across all of 2024 [4]. NetScout recorded over 8 million attacks in the first half of 2025 alone, with several campaigns peaking at 3.12 Tbps, demonstrating how terabit-scale assaults have become routine rather than exceptional [5]. The largest known case to date reached 7.3 Tbps and pushed 37.4 TB of data in under a minute, setting a new benchmark for the destructive potential of DDoS attacks [6].

To complement industry surveys, institutional reports offer a broader perspective. ENISA highlights that threats against availability (DDoS) remained the top-ranked threats throughout 2024, reinforcing the criticality of DDoS in the modern threat landscape. Furthermore, ENISA stresses that large-scale outages caused by denial-of-service operations threaten not only enterprises but also national infrastructures, positioning DDoS as a systemic cyber risk [7].

The proliferation of massive botnets-networks comprising millions of compromised devices, amplifies attackers' ability to sustain prolonged, distributed DDoS campaigns. For instance, the New Jersey Cybersecurity & Communications Integration Cell (NJCCIC) reported that the Eleven11bot botnet has rapidly expanded to involve over 86,000 compromised devices, primarily targeting vulnerable IoT systems such as routers, DVRs, and cameras, which are then leveraged for large-scale DDoS campaigns [8].

Moreover, DDoS attacks have increasingly been adopted by politically and ideologically motivated groups, often fortified by automation and AI-powered tools. According to Europol, the hacktivist group NoName057, active since March 2022, has executed daily campaigns against over 3,700 targets, predominantly government and public-sector websites in Europe, with attack intensity closely aligned to geopolitical developments [9].

These observations confirm that DDoS attacks are not just transient network disruptions but systemic threats with far-reaching consequences. Their rising frequency, scale, and sophistication turn them into a persistent risk for both enterprises and critical infrastructures. Traditional mitigation strategies often struggle to keep pace with dynamic attack patterns, leaving organizations exposed to escalating technical and economic damages. This growing gap between evolving threats and static defenses underscores the need for adaptive, data-driven mechanisms capable of responding in real time within the network itself.

Recently, research on DDoS attack detection has increasingly focused on how to leverage In-Network Machine Learning (IN-ML), enabled by the P4 language, to identify attacks directly within network switches. P4 allows programmers to define the behavior of the programmable data plane (PDP), making it possible to customize how packets are parsed, processed, and forwarded at line rate. This enables switches to implement not only forwarding policies but also monitoring and security functions directly in hardware. The goal is to handle massive floods without becoming a bottleneck, something that CPU/GPU-based IDS systems often struggle with, and to reduce the attack surface by preventing malicious traffic from ever reaching the servers, all while processing packets at Tbps scale.

However, this approach comes at a cost. To guarantee line-rate forwarding, only a limited set of operations is supported, which prevents the direct implementation of complex ML models. Training ML or RL algorithms requires heavy computations and large datasets; as a result, current PDPs can realistically support only inference, while training must be performed offline or in the control plane. Furthermore, due to simplified detection logic and constrained feature extraction, in-network ML may face precision/recall trade-offs.

Given this context, in this thesis we answer these research questions through the following contributions:

- **RQ1** Can a programmable data plane learn and update a DDoS-mitigation policy online at line rate without control-plane involvement?
- **C1** The proposed system can adapt in real time to evolving traffic patterns and identify malicious flows at line rate.
- **RQ2** What ML algorithm fits P4/RMT limits while remaining effective?
- **C2** Algorithm selection via literature review: an analysis of selected works mapping candidate ML/RL designs to P4/RMT limitations, with explicit pros and cons to guide the choice.
- **RQ3** Which line-rate features expose amplification behavior for learning?

- **C3** DDoS-aware telemetry and reward: line-rate asymmetry features and an in-switch reward computation tailored to amplification.
- **RQ4** How does in-switch learning compare to static filtering policies in terms of mitigation effectiveness and decision behavior?
- **C4** Prototype & evaluation: a working P4 prototype and experimental comparison with a static threshold-based filtering, highlighting the benefits of adaptive mitigation decisions based on temporal traffic context.

1.2 Thesis Structure

The remainder of this thesis is structured as follows. chapter 2 introduces the background necessary to understand the problem addressed in this work, providing an overview of Distributed Denial of Service attacks and presenting the concepts of programmable data planes and in-network machine learning, which constitute the technological foundation of the proposed approach.

Building upon this background, chapter 3 reviews the fundamentals of reinforcement learning and discusses existing approaches that apply RL techniques to network security and DDoS mitigation.

The objectives of this thesis and the research questions that guide the study are presented in chapter 4. Subsequently, chapter 5 describes the proposed methodology, including the system model, the formulation of the problem as a Markov Decision Process, and the reinforcement learning framework adopted for mitigating amplification-based attacks.

The practical realization of the proposed solution is detailed in chapter 6, which describes the implementation of the system, including the design of the P4 data-plane components and the control-plane logic that supports the learning process. The experimental evaluation of the proposed approach is presented in chapter 7, where the experimental setup is described and the results obtained under different attack scenarios are analyzed and discussed.

Finally, chapter 8 concludes the thesis by summarizing the main findings of this work, discussing the limitations of the proposed approach, and outlining possible directions for future research.

Chapter 2

DDoS and PDP

2.1 Distributed Denial of Service Attacks

2.1.1 Definition and Classification

According to the NIST [10], a Denial of Service (DoS) attack can be described as the prevention of authorized access to resources or the delaying of time-critical operations. As suggested by this definition, these attacks do not necessarily damage data directly; the attacker's objective is to block or degrade access to a computing resource or service. DoS attacks can be classified into five categories based on the attacked protocol level [1]:

- **Network Device level**: Attacks that exploit software bugs or hardware limitations of network devices. An example is the *Cisco IOS HTTP Server Vulnerability (2001)*, where crafted HTTP requests could crash the router or execute arbitrary code.
- **OS level**: These attacks target the way operating systems implement protocols. A notable example is the *Teardrop Attack*, which exploited flaws in how some systems handled fragmented IP packets, causing crashes or reboots.
- **Application level**: Focused on vulnerabilities in network applications or exploiting algorithmic complexity to exhaust resources. The *Slowloris Attack* exemplifies this by keeping many HTTP connections half-open, exhausting server resources with minimal bandwidth.
- **Data flood**: Aims to overwhelm bandwidth or host resources by sending massive volumes of traffic, often with spoofed source addresses. A common case is the *UDP Flood*, where attackers send large numbers of UDP packets to random ports, consuming bandwidth and CPU cycles.
- **Protocol feature attack**: Leverage inherent features of network protocols. A classic example is the *TCP SYN Flood*, which exploits the three-way handshake by sending numerous SYN requests with spoofed IPs, leaving the server with many half-open connections.

2.1.2 Characteristics and Phases

A denial-of-service technique that relies on a large number of compromised hosts to carry out the attack is known as a Distributed Denial of Service (DDoS). It represents one of the most sophisticated and dangerous forms of DoS attacks. DDoS attacks derive much of their strength from the very foundations of the Internet's design. The network was originally built with functionality as the primary goal, leaving security considerations in the background and thus exposing a range of weaknesses that adversaries can exploit. A first issue is the high level of interdependence between systems: even a well-secured host can fall victim to a DDoS attack if other parts of the global Internet infrastructure are compromised. Another limiting factor lies in the finite nature of online resources, since no host possesses infinite capacity and sustained pressure from multiple sources will inevitably exhaust available bandwidth or processing power. The imbalance of "many against few" means that, when attackers collectively command greater resources than their target, the likelihood of overwhelming it becomes almost certain. Furthermore, the distribution of intelligence and resources across the network plays a role: decision-making capabilities largely reside in end hosts, while intermediate nodes are optimized to provide high-throughput communication. This separation allows attackers to misuse the considerable bandwidth of unsuspecting networks to generate massive traffic volumes and direct them against a chosen victim.

A DDoS attack typically involves four main components. At the top of the hierarchy shown in Figure 2.1 is the attacker, who initiates and coordinates the operation. Below are the handlers (or masters), compromised systems equipped with specialized software that enables them to control large groups of subordinate machines. These subordinates, known as agents or zombies, are infected hosts responsible for generating the malicious traffic directed at the target. To maximize effectiveness and reduce the chance of countermeasures, agents are usually located outside both the attacker's and the victim's networks. Finally, at the bottom of the chain is the victim, the host or service targeted by the attack.

The execution of a DDoS attack follows a series of preparatory and operational phases [1]:

1. **Agent Selection**: The attacker first identifies suitable machines to be turned into agents. These systems must present exploitable vulnerabilities and possess enough computational and bandwidth resources to generate substantial traffic. While this stage was once performed manually, it has since been automated with scanning tools.

2. **Compromise**: Once selected, the machines are infected by exploiting software flaws or configuration weaknesses. Malicious code is installed and often concealed to prevent detection or removal. Over time, worms such as Ramen and Code Red automated this step, enabling large-scale infections. Importantly, the agents consume only minimal resources when idle, so their owners typically remain unaware that their machines are being used in an attack.

3. **Command and Control**: The attacker maintains communication with one or more handlers to manage the botnet. Through these channels, the attacker

can check which agents are active, coordinate the timing of attacks, or distribute updates to the malware. Communication between attacker, handlers, and agents may use various protocols, including TCP, UDP, or ICMP.

4. **Attack Execution** In the final phase, the attacker instructs the agents to launch the assault. Parameters such as the target address, the duration of the attack, and specific packet characteristics (e.g., type, size, TTL, port numbers) can all be configured. By varying these properties, attackers can evade simple detection and filtering mechanisms, making the attack more difficult to mitigate.

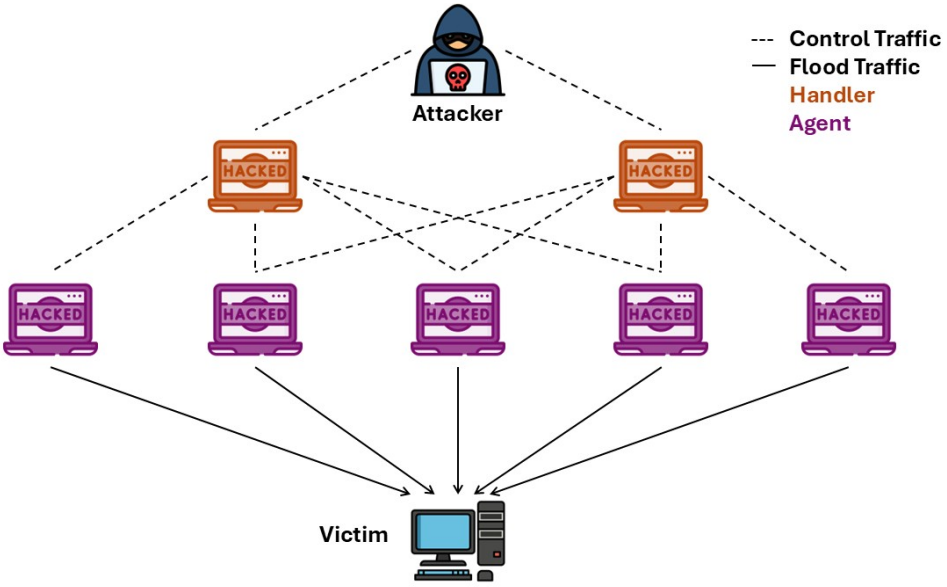

Figure 2.1: Illustration of the threat model in Distributed Denial of Service (DDoS) attacks, Figure adapted from [1].

2.1.3 Flooding and Amplification Techniques

Within the wide range of Distributed Denial of Service techniques, two categories stand out as particularly relevant for this thesis: flooding and amplification attacks. These strategies represent distinct ways of overwhelming a target, either by directly saturating its resources with massive traffic volumes or by leveraging intermediate systems to magnify the attack's impact. In the following, both approaches are described in detail, highlighting their mechanisms and emphasizing the fundamental differences between them.

Flooding Attacks In flooding attacks, compromised machines (zombies) overwhelm a target by transmitting massive amounts of IP traffic, saturating its bandwidth. The consequences of such traffic streams can range from performance degradation and system crashes to the complete exhaustion of network capacity. Common examples of this category are UDP flood, ICMP flood, and TCP SYN flood attacks.

A UDP flood occurs when a victim is bombarded with a high volume of UDP packets, which quickly consumes the available bandwidth and prevents legitimate requests from being processed. In distributed versions of this attack, the packets may target either random or specific ports on the victim's system. When a UDP packet arrives at a port where no service is running, the victim responds by generating an ICMP "destination unreachable" message addressed to the (spoofed) source. At scale, this exchange can overwhelm the victim's resources, and by using spoofed source addresses the attacker also hides the origin of the malicious traffic.

An ICMP flood exploits the Internet Control Message Protocol, which is commonly used to test connectivity. During such an attack, agents send large numbers of ICMP echo requests ("pings") to the target, each requiring a response. This barrage of packets consumes the victim's network bandwidth and processing resources, often leading to service disruption. As with UDP floods, IP spoofing is typically used to disguise the actual source of the attack.

A TCP SYN flood takes advantage of the three-way handshake mechanism used to establish TCP connections. The attacker sends a large number of SYN packets with spoofed source addresses to the victim. Each of these packets prompts the server to allocate resources and send back a SYN-ACK, but the final ACK from the client never arrives. As a result, the server's backlog queue fills with half-open connections, eventually preventing it from accepting legitimate requests.

Amplification Attacks DDoS attacks have evolved beyond the traditional command-and-control model into amplification attacks, where adversaries exploit legitimate third-party services or protocols to multiply the traffic directed toward a victim. The basic principle lies in sending a request that is either very small in size or limited in number but crafted in such a way that the responses generated are much larger or more numerous. By spoofing the victim's IP address as the source of the requests, the attacker ensures that all amplified responses are directed at the target rather than at themselves, which makes both mitigation and traceback significantly more difficult. This combination of efficiency and anonymity is further reinforced by the fact that most protocols vulnerable to amplification-based DDoS attacks are UDP-based, as UDP lacks a handshake mechanism to verify the legitimacy of the sender's address. Together, these characteristics make amplification attacks highly effective and attractive to adversaries.

Within this category, researchers distinguish between flow multiplication and payload magnification [11]. Flow multiplication is achieved when a single malicious request triggers many response packets. Early examples include the Smurf attack, where ICMP Echo Requests are sent to the broadcast address of a network, causing every host in that network to respond with an Echo Reply to the spoofed victim. Similarly, the Fraggle attack abuses the UDP echo and chargen services in the same way, generating numerous responses for each spoofed request. The common factor in these attacks is the multiplication of the number of flows: a limited set of spoofed packets produces a much larger set of responses, quickly overwhelming the victim's resources.

Payload magnification, on the other hand, does not increase the number of flows but rather exploits services that return disproportionately large responses relative

to the original request size. To quantify the extent of amplification, Rossow [12] defines the *Bandwidth Amplification Factor (BAF)* as the ratio between the size of the response sent by the amplifier and the size of the request generated by the attacker:

$$BAF = \frac{\text{len(UDP payload)}_{\text{amplifier}\rightarrow\text{victim}}}{\text{len(UDP payload)}_{\text{attacker}\rightarrow\text{amplifier}}}$$

A higher BAF means that even a small request can trigger a disproportionately large response, making the protocol particularly appealing for abuse in amplification attacks. Additionally, the authors define the *Packet Amplification Factor (PAF)* as a measure of how many IP packets an amplifier emits in response to a single request:

$$PAF = \frac{\text{Number of packets}_{\text{amplifier}\rightarrow\text{victim}}}{\text{Number of packets}_{\text{attacker}\rightarrow\text{amplifier}}}$$

Rossow [12] provides a systematic analysis of a broad set of such protocols, with the key results summarized in Table 2.1.

A classic case is the DNS amplification attack, where a small query, such as an ANY request, can generate a response tens of times larger, with amplification factors ranging from 28x to over 50x. Another notorious example is the NTP amplification attack, which abuses the monlist command to return extensive information about recent clients, leading to amplification factors as high as 556x. More recently, the abuse of the Memcached protocol has shown how payload magnification can be taken to extremes, with amplification factors exceeding 50,000x, making it one of the most powerful amplification vectors observed in practice.

In both cases, the outcome is the same: massive volumes of unsolicited traffic overwhelming the victim's bandwidth and computational resources, yet the underlying mechanisms differ, underscoring the need for tailored defense strategies against each type of abuse.

Table 2.1: Bandwidth and Packet amplification factors of common protocols, as reported by Rossow (2014).

Protocol	**Bandwidth amplification factor**	**Packet amplification factor**	**Vulnerable command**
DNS	28 to 54	1.32	–
NTP	556.9	10.61	–
SNMPv2	6.3	1.00	GetBulk request
NetBIOS	3.8	1.00	Name resolution
SSDP	30.8	9.92	SEARCH request
CharGEN	358.8	1.00	Character generation request
QOTD	140.3	1.00	Quote request
BitTorrent	3.8	1.58	File search
Kad	16.3	1.00	Peer list exchange
Quake Network Protocol	63.9	1.01	Server info exchange
Steam Protocol	5.5	1.12	Server info exchange
Multicast DNS (mDNS)	2 to 10	–	Unicast query
RIPv1	131.24	–	Malformed request
Portmap (RPCbind)	7 to 28	–	Malformed request
LDAP	46 to 55	–	–
CLDAP	56 to 70	–	–
TFTP	60	–	–
Memcached	10,000 to 51,000	–	–

2.2 Programmable Data Planes and In-Network Machine Learning

2.2.1 Programmable Data Plane Architecture

The idea of the programmable data plane arose as the next step after software-defined networking (SDN). Traditional SDN separated the control plane from the data plane, but its flexibility was mostly confined to adjusting forwarding and routing rules, while the underlying data plane remained largely static. With the advent of programmable data planes, packet-processing logic itself can now be redefined, allowing network functions to be executed directly within the forwarding path. In this way, operators can run applications in-network at line rate, adapting in real time to the dynamic requirements of modern infrastructures. Packet streams can be selectively processed on the fly as they traverse the network, without needing to offload tasks to external systems.

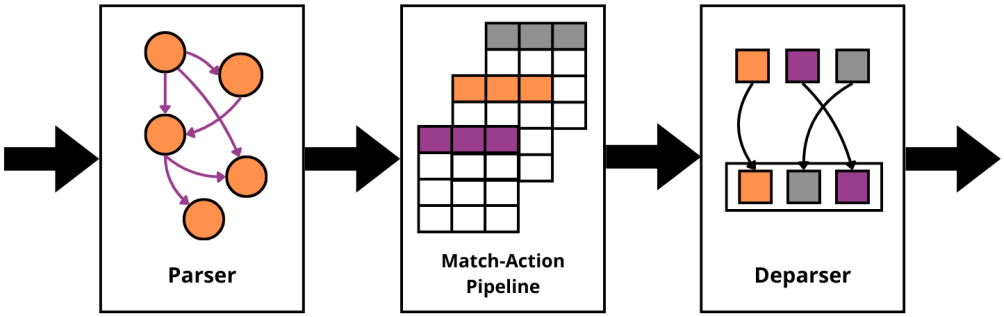

Figure 2.2: Schematic representation of the Protocol-Independent Switch Architecture (PISA), Figure adapted from [2].

Programmable devices rely on specialized architectures designed for flexible packet handling. A central model is the Protocol-Independent Switch Architecture (PISA) that, unlike fixed-function hardware, enables customization of packet processing without depending on vendor-specific protocol support. Its pipeline consists of three key elements: the parser is the entry point of the pipeline. It reads the incoming bitstream sequentially and extracts protocol fields based on a finite state machine defined in the P4 program. At each stage, the parser identifies the next header type (e.g., Ethernet, IPv4, UDP) and populates the *Packet Header Vector* (PHV). The PHV contains both protocol-specific fields, such as IP addresses or port numbers, and intrinsic metadata, including ingress or egress ports, timestamps, or packet length. This structured representation provides the foundation for the programmable logic that follows.

The match-action stages form the core of the programmable pipeline. Selected

fields from the PHV are used as keys to perform lookups in a sequence of tables. These tables support various types of matches, such as exact match, ternary match with wildcards, and longest prefix match. Once a rule is matched, the corresponding action is executed. Actions may include simple operations, like header field modifications or TTL decrements, but also more advanced tasks, such as updating stateful registers, incrementing counters, or executing custom logic. Since the pipeline is composed of multiple stages, complex decision-making can be realized by chaining match-action tables. The use of registers further enables stateful packet processing, allowing the data plane to maintain per-flow or global state directly at line rate.

Finally, the deparser reconstructs the packet for transmission. It serializes the modified PHV back into packet headers, ensuring correct ordering and alignment, and appends the original payload. This stage makes it possible to output packets with altered headers, such as updated addresses, encapsulations for tunneling, or additional telemetry metadata. By controlling the deparser, operators can introduce new protocols or embed diagnostic information while preserving payload integrity, thus enabling advanced functions such as in-band telemetry and custom header insertion.

2.2.2 The P4 Programming Language

A fundamental component enabling programmable data planes is the P4 language (*Programming Protocol-Independent Packet Processors*), which has rapidly established itself as the de facto standard for programming the forwarding behavior of modern network devices. P4 is a domain-specific language explicitly designed for packet processing, allowing developers to define how packets are parsed, which fields are matched, and which actions are applied. Its main strength lies in protocol independence: new headers can be introduced or obsolete ones removed simply by updating the program, without requiring hardware redesign or vendor intervention. Moreover, P4 provides a clean separation between data and control planes, while still enabling their interaction. The control plane can dynamically populate or modify match-action tables at runtime, update counters, or retrieve telemetry data, thus complementing the static logic expressed in the P4 program.

From a development perspective, the *P4 language and core library* provide the common abstractions and constructs defined by the P4.org community, ensuring portability across targets. At the same time, each vendor supplies an *architecture definition*, an abstract model of the programmable device such as a switch, router, or NIC, along with specific *extern libraries* that expose hardware-dependent functions, enabling interaction with device-specific capabilities like parsers, match-action pipelines, or deparsers. P4 programs must be compiled to run on specific hardware or software targets. The reference compiler, known as *p4c*, has been developed by the P4 community and extended by hardware vendors to support their products. Compilation transforms a P4 program into a binary compatible with the architecture of the selected target. The most widely adopted language version today is P4_16.

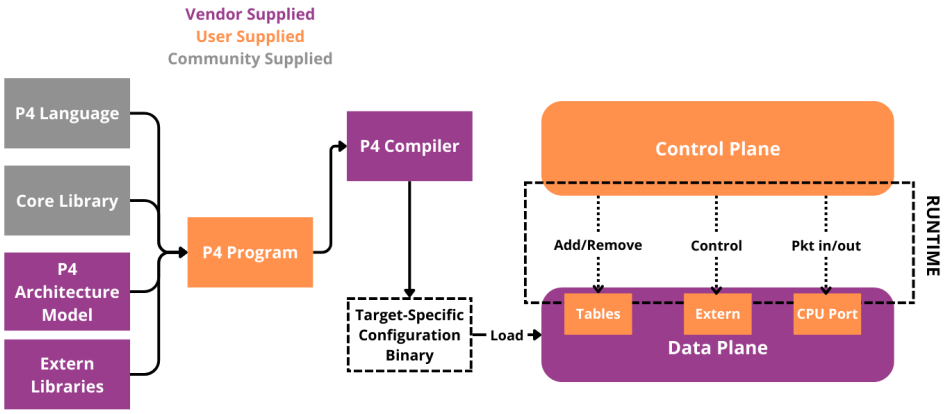

Figure 2.3: Overview of the P4 programming workflow, Figure adapted from [2].

In terms of execution environments, P4 is remarkably versatile, supporting a wide variety of programmable targets. At the hardware level, it is implemented on programmable switch ASICs such as Intel Tofino or Cisco Silicon-One, which offer high-performance line-rate packet processing. Beyond switches, P4 is also supported on *SmartNICs*, *Data Processing Units* (DPUs), and *Infrastructure Processing Units* (IPUs), such as NVIDIA BlueField, Intel IPU and Netronome NFP, which combine network programmability with offload capabilities. P4 can also be mapped onto *Field-Programmable Gate Arrays* (FPGAs), providing a flexible platform for custom packet processing, as demonstrated by research prototypes using NetFPGA or commercial devices like AMD Alveo. Finally, P4 has been ported to software switches, including *BMv2*, the open-source behavioral model widely used in research and testing for experimentation.

Crucial to this ecosystem are the abstract architectures that define the programming model for a given target. The *Portable Switch Architecture* (PSA) represents the canonical model for programmable switches, while the *Portable NIC Architecture* (PNA) extends similar concepts to network interface cards. Other commonly used abstractions include the *v1model*, often employed in academic settings, and vendor-specific architectures such as Intel's *Tofino Native Architecture* (TNA). These models define the structure of the pipeline: parser, match-action stages, and deparser and the interfaces exposed to the control plane. By combining architecture-specific flexibility with the general-purpose abstractions of P4, operators can tailor packet processing to their needs while ensuring portability across different platforms.

When it comes to the target of our interest, the experiments presented in this work rely on the **BMv2** behavioral model, which serves as the open-source software reference switch for P4. Specifically, the **v1model** architecture was adopted, as it is the default and most widely used abstraction for BMv2 in academic and research contexts. This choice provides a stable and well-documented environment to implement and test programmable data plane functionalities, making it particularly suitable for prototyping and experimental validation. The *v1model* architecture and its characteristics will be discussed in greater detail in the following chapters, where its role in supporting our implementation is examined more closely.

2.2.3 In-Network Machine Learning Paradigm

The rise of programmable network devices has introduced an entirely new paradigm, where machine learning functionality can be deployed directly within the network to address modern requirements. This approach is motivated by the shortcomings of traditional designs, in which machine learning models are typically executed on servers or external accelerators. While such systems have been widely used for tasks like traffic classification, anomaly detection, and automated configuration, they inevitably introduce additional latency and place extra computational load on central resources.

By contrast, in-network machine learning shifts part of this workload into the data plane itself. Processing decisions closer to where traffic actually flows reduces latency, a property that is particularly advantageous in scenarios demanding immediate responsiveness, such as intrusion detection [13] or congestion control [14]. At the same time, offloading portions of analysis to programmable switches or network interface cards alleviates the strain on servers and accelerators, freeing them to handle other workloads. In addition, in-network ML can support more efficient distributed learning: by aggregating gradients or parameters directly within the network, communication bottlenecks that often hinder large-scale training can be mitigated. Beyond performance gains, embedding ML capabilities in the data plane also enhances network adaptability, enabling infrastructures to dynamically adjust forwarding strategies, balance traffic loads, and enforce security policies without external intervention.

2.2.4 Security-Oriented Case Studies in In-Network ML

Taken together, these advantages underscore the potential of in-network machine learning to transform current infrastructures into faster, lighter, and more intelligent networks, better suited to the demands of cloud services, edge computing, and security-critical applications.

For the purposes of this thesis, it is particularly relevant to focus on research that applies in-network machine learning to security-related challenges. A further aspect of interest lies in how these approaches handle the training phase of machine learning models. Since training in-network is often constrained by the limited computational resources of programmable devices, different strategies have been proposed, such as offline training with subsequent deployment of inference logic in the data plane, or hybrid schemes that leverage both control and data planes.

Examining these design choices provides valuable insights into the feasibility and trade-offs of designing a DDoS mitigation system that performs its training directly in the data plane. Achieving this would represent a significant step forward, enabling networks to move beyond simply hosting inference tasks and evolve into fully autonomous learning systems, which is the central objective of this thesis.

These two works that will be presented next are a representative example of the deployment of inference logic directly within the data plane, showing how different machine learning models can be mapped onto programmable switches to operate at line rate. In the first case, *P4-NIDS* [13] implements a network intrusion detection

system by encoding decision-tree classifiers into P4 tables and actions. Here, the inference logic is mapped onto the match–action pipeline: the switch parses packet headers, extracts features such as protocol identifiers and port numbers, and evaluates a pre-trained decision tree encoded as a cascade of P4 if–else statements. These predicates read per-flow intermediate state from registers and, upon a match, set a "malicious" flag in a dedicated register for enforcement. This allows the system to detect intrusions inline, without leaving the forwarding path, ensuring both minimal latency and real-time responsiveness.

The second work, *In-Line Any-Depth Deep Neural Networks Using P4 Switches* [15], demonstrates how deep neural networks can also be realized within the data plane. Instead of relying on precomputed trees, this approach maps the inference operations of Deep Neural Networks onto the PISA pipeline by replacing matrix and activation computations with deterministic table matches over concatenated, integer-encoded features across the stages. Lookup tables approximate non-linear functions, while registers maintain partial results across pipeline stages. The authors design the system to allow for arbitrary network depth, chaining together stages to emulate successive layers of the neural network. This design makes per-packet decisions directly inside the switch, and the authors show two concrete uses: a DDoS mitigator and an IoT malicious-activity classifier.

Together, these examples illustrate the feasibility and versatility of embedding inference logic directly into programmable switches, but both of these data-plane ML deployments inevitably surface the structural limits of today's programmable data planes.

While these works demonstrate the feasibility of embedding inference logic directly into the data plane, they also expose the inherent limitations of current programmable data plane devices. Both P4-NIDS and the LUT-distilled DNN system rely on careful reformulation of learning algorithms to fit the restricted set of operations supported by Reconfigurable Match–Action Tables (RMT). On such PDP devices, computation and memory are evenly partitioned across pipeline stages; each stage may only use its own local resources, a register can be accessed at most once per stage, and operations like floating-point arithmetic, multiplication, or large-vector processing are not supported at line rate.

As a workaround for the lack of native ML primitives in the data plane, and to extend PDPs' supported operations, both systems adopt the most straightforward tactic: lookup tables. By precomputing the possible calculation requests and storing them in memory, complex operations are reduced to simple lookups, enabling RMT-based PDPs to support computations such as multiplication and logarithms.

These choices, however, entail concrete downsides: integer-only encodings and the absence of multiply–accumulate operations reduce numerical fidelity relative to floating-point pipelines (quantization may introduce accuracy loss), while LUTs face exponential memory growth with input bit-width, forcing hierarchical decompositions and tightly bounded feature sets. On the *P4-NIDS* side, only a narrow slice of pre-trained, lightweight models is practically portable to the switch, ruling out richer architectures and any in-switch training.

Stepping back from the two case studies, I now discuss general programmable data-plane constraints and design trade-offs as synthesized by the Programmable

Data Plane Intelligence survey [16]. Additional challenges stem from latency budgets and stage restrictions (fewer stages mean fewer resources), and from memory organization: registers and tables are distributed across stages in RMT and hierarchically arranged in Network Processors, making it difficult to deploy large feature sets or models that require significant shared state. Concretely, each match–action stage adds a few nanoseconds of delay; to keep latency low the pipeline comprises only tens of stages, so total per-packet overhead is on the order of a few hundred nanoseconds, still acceptable relative to µs baseline delays in non-programmable data-center switches, but it tightens the resource budget available to programs. PDPs may seem to have plenty of memory, but their design limits how much of it you can actually use: on network processors, memory is hierarchical and demands careful placement to avoid slow accesses; on RMT switches, memory and compute are evenly partitioned across stages and each stage can access only its local memory. To cope with these constraints, tailor-made data structures and compact feature encodings are used to keep state within stage-local budgets. When a program needs more memory than a single stage provides, it must consume additional stages solely to extend memory access, wasting pipeline capacity even if no extra computation is performed. These factors explain the persistent gap between ML requirements and PDP capabilities.

Chapter 3

Reinforcement Learning

Motivated by the limits exposed in the previous chapter 2, we looked beyond security for ideas on whether training could be brought into the data plane; this led us to QCMP [17], a load-balancing system that implements in-network reinforcement learning (Q-learning) on programmable switches.

3.1 Foundations of Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning in which an *agent* learns how to act by interacting with an *environment*, aiming to maximise the total *reward* gathered over time. In contrast to supervised learning, where labelled examples specify the correct outputs, RL provides no explicit instruction about which action is "right" in each situation. Instead, learning unfolds through trial and error: the agent tries actions and receives feedback in the form of rewards or penalties based on the consequences.

A distinctive aspect of RL is its treatment of delayed rewards. An action affects not only the immediate payoff but also the future state of the environment and the rewards that come later. This creates a natural need to strike a balance between exploration, meaning trying unfamiliar actions to discover better strategies, and exploitation, meaning choosing actions already known to produce high returns. Handling this tension is a central challenge specific to RL and does not appear in purely supervised or unsupervised learning.

The RL problem is typically framed as optimal control over a *Markov Decision Process* (MDP), potentially under partial observability. The agent perceives (possibly imperfect) state information, selects actions that shape the environment's evolution, and optimises an objective expressed via rewards [18].

The agent's ability to continuously adjust its behavior in response to the outcomes of its past choices makes reinforcement learning particularly suitable for dynamic networking scenarios. In such environments, traffic conditions can shift rapidly, and static policies are unable to react in time. RL, by contrast, adapts as the system evolves, ensuring that decision-making remains aligned with the current state of the network.

The main components of this sequential decision-making framework are:

- **Agent**: the decision-making component that acts based on information received from the environment. Examples include a robot learning to navigate, a chess-playing program, or an autonomous driving system.
- **Environment**: the surrounding context with which the agent interacts. It provides the current state and evolves in response to the agent's actions.
- **Action**: the decision taken by the agent in a particular state. Actions determine how the environment changes and influence subsequent rewards.
- **Reward signal**: a numerical feedback provided by the environment after each action, indicating the immediate benefit or cost of that decision. It defines short-term desirability and serves as the main criterion for updating the agent's policy.
- **Policy**: the strategy mapping states to actions. It may be deterministic, always selecting the same action in the same state, or stochastic, assigning probabilities to possible actions. The policy fully defines the agent's behaviour.
- **Value function**: an estimate of the long-term return from a state or a state–action pair, expressed in terms of expected future rewards. While rewards capture only the immediate outcome, the value function guides the agent toward actions that maximise cumulative gains. It is central to most RL algorithms.
- **Model of the environment**: an internal representation used to predict state transitions and rewards for given actions. Algorithms that exploit such models are referred to as model-based and can plan by simulating future scenarios, whereas model-free methods rely purely on trial and error.
- **RL decision loop**: learning follows the iterative cycle in Figure 3.1

  $$s_t \rightarrow a_t \rightarrow r_{t+1},\ s_{t+1},$$

  which generates sequences of state–action–reward–next state tuples (s, a, r, s'). From these, the agent learns to maximise the expected return, typically expressed as the discounted sum of future rewards.
- **Episodic and continuing tasks**: reinforcement learning problems can be episodic, with a clear initial and terminal state, or continuing, where interactions proceed indefinitely without a natural end.
- **Exploration vs Exploitation**: A central challenge in reinforcement learning is the trade-off between *exploration* and *exploitation*. Exploitation refers to the agent's tendency to select the action that appears to yield the highest reward based on its current knowledge. This strategy maximizes immediate returns but risks overlooking better alternatives. Exploration, on the other

hand, involves deliberately selecting actions that are not currently estimated as optimal, with the aim of acquiring new information about their long-term potential. While exploration may temporarily reduce rewards, it is essential for discovering actions that could yield higher cumulative returns in the future. The dilemma arises because pursuing only exploitation can trap the agent in suboptimal behavior, whereas relying solely on exploration prevents consistent reward maximization. The ϵ-*greedy* strategy addresses this by introducing a probabilistic balance: with probability $1-\epsilon$ the agent exploits its current knowledge by choosing the best-known action, while with probability ϵ it explores by selecting an action at random. In this way, ϵ-greedy ensures that all actions are eventually sampled, while still favoring those estimated to be most rewarding, striking a practical compromise between short-term gain and long-term learning.

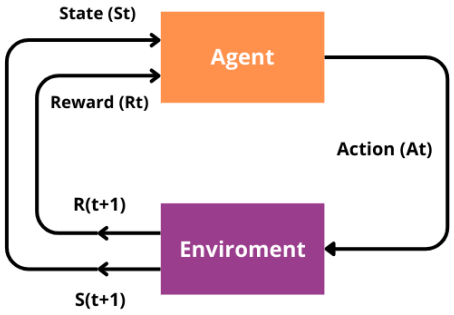

Figure 3.1: Graphical representation of the reinforcement learning decision loop, showing the interaction between state, action, reward, and next state.

In reinforcement learning, two fundamental classes of methods are *value-based* and *policy-based*. Value-based approaches focus on estimating a value function, such as the action-value function $Q(s,a)$, which represents the expected return of taking action a in state s and then following a given policy. The optimal action is obtained by selecting the one that maximizes this value:

$$a^* = \arg\max_a Q(s,a).$$

Policy-based methods, instead, directly parameterize the policy $\pi_\theta(a|s)$ and optimize its parameters θ through gradient ascent on the expected return. This leads to the policy gradient theorem:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s)\, Q^{\pi_\theta}(s,a)\right],$$

which provides a practical way to adjust the policy parameters in the direction that increases the expected performance.

Another key difference lies in the way value estimates are updated, especially with respect to the influence of bias and variance on these estimates.

Monte Carlo (MC) methods estimate returns by waiting until an episode terminates and then using the empirical return to update value estimates. For a state

s_t, the MC target is the full return

$$G_t \;=\; \sum_{k=0}^{T-t-1} \gamma^k \, r_{t+1+k},$$

and a typical state-value update is

$$V(s_t) \;\leftarrow\; V(s_t) \;+\; \alpha\big(G_t - V(s_t)\big).$$

These updates are *unbiased* with respect to the true value under the generating policy, but can exhibit high variance because G_t depends on all subsequent rewards in the episode.

Temporal-Difference (TD) methods, by contrast, update value estimates after each step using *bootstrapped* targets that incorporate the current value estimate (i.e., they rely on the learner's own present estimate rather than the full empirical return, trading some bias for reduced variance). In TD(0), the one-step target for s_t is

$$r_{t+1} + \gamma \, V(s_{t+1}),$$

yielding the update

$$V(s_t) \;\leftarrow\; V(s_t) \;+\; \alpha\Big(r_{t+1} + \gamma \, V(s_{t+1}) - V(s_t)\Big).$$

TD learns online and incrementally, requiring no episode termination, and typically achieves lower variance thanks to bootstrapping, at the cost of introducing bias through the use of $V(s_{t+1})$.

In summary, Monte Carlo targets the full return and is therefore unbiased but often high variance. TD uses bootstrapped, one-step targets that reduce variance while introducing bias. As the value estimates become more accurate, the bias in TD updates diminishes, leading to a favorable balance for incremental, online learning.

3.2 QCMP: In-Switch Q-Learning

After introducing the fundamentals of reinforcement learning, we now turn to QCMP [17], one of the first works to *use the limited resources of a programmable data plane to carry out agent training* on switch hardware. Though it addresses load balancing rather than DDoS mitigation, its key novelty is real-time reinforcement learning that lets policies adapt rapidly to shifting traffic. Unlike conventional schemes such as ECMP, which rely on static hash-based distribution, QCMP employs Q-learning to iteratively refine path weights across multiple next hops. Q-learning is especially suitable here because it dispenses with an explicit model of the (complex and unpredictable) network environment, learning optimal weight adjustments through trial and error. Using aggregated queue lengths as state, actions that adjust per-path weights, and a reward that jointly minimizes weighted queue occupancy and inter-path imbalance, QCMP updates its Q-table at line rate, continually improving the policy as traffic evolves, under strict programmable data plane constraints. For these reasons, a focused analysis of Q-learning and how it has been implemented in P4 is essential to the objectives of this thesis.

3.2.1 Q-Learning Algorithm

Q-learning is a model-free, value-based reinforcement learning algorithm designed to learn the optimal action-selection policy through direct interaction with the environment. The agent maintains a *Q-table*, where each entry $Q(s, a)$ estimates the expected cumulative reward of taking action a in state s and thereafter following the current policy. At each step, the agent observes its current state s, selects an action a (often using an ϵ-greedy strategy to balance exploration and exploitation), receives a reward r, and transitions to a new state s'. The Q-value is then updated according to the temporal-difference rule:

$$Q(s, a) \leftarrow Q(s, a) + \alpha \Big[r + \gamma \max_{a'} Q(s', a') - Q(s, a) \Big],$$

where:

- $Q(s, a)$: current estimate of the action-value for state s and action a;
- α: learning rate, controlling how much new information overrides the old estimate;
- r: immediate reward received after performing action a in state s;
- γ: discount factor, weighting the importance of future rewards relative to immediate ones;
- $\max_{a'} Q(s', a')$: the maximum estimated action-value achievable from the next state s';
- $r + \gamma \max_{a'} Q(s', a') - Q(s, a)$: the temporal-difference error, which quantifies the gap between the current estimate and the improved estimate based on new experience.

Over repeated interactions, the estimates in the Q-table converge toward the true action values, enabling the agent to approximate the optimal policy without requiring a model of the environment.

3.2.2 Register-Based Q-Learning in P4

Q-learning is particularly suitable for mapping into the data plane because its tabular value-iteration structure aligns with the primitives of the PISA architecture. The Q-table can be stored in `register` arrays, while state and action identifiers are encoded as indices. This design allows Q-values to be updated directly within the pipeline at line rate, using read-modify-write operations on registers and precomputed arithmetic via match-action tables. In QCMP, the workflow of register-based Q-learning is organized as follows [17]:

1. **State and reward extraction:** When a packet arrives, the switch derives the new state s' and the immediate reward r from network telemetry, while storing the previous state s and action a in registers.

2. **Next-state evaluation:** The Q-values $Q(s', *)$ corresponding to the new state are read from the Q-table registers.

3. **Action selection:** The switch selects the next action a' using an ϵ-greedy policy, balancing exploration and exploitation.

4. **Register update:** The registers holding $(s, a, Q(s, a))$ are updated with the new triplet $(s', a', Q(s', a'))$.

5. **Q-value update:** The temporal-difference update

$$Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma \max_{a'} Q(s', a') - Q(s, a)\big]$$

   is computed in the pipeline. Since multiplications are not natively supported, the factors α and γ are precomputed and stored in lookup tables, allowing the update to be realized with additions and table lookups.

6. **Action execution:** Finally, the chosen action a' is applied to the packet forwarding decision, closing the loop of learning directly inside the data plane.

This register-based realization allows QCMP to continuously adapt its policy in real time, exploiting the operations natively supported by P4-programmable switches without requiring control-plane intervention.

3.3 RL-based DDoS Defenses

After identifying Q-learning as a viable algorithmic primitive for the data plane, we broadened our investigation to assess whether *reinforcement learning* can serve as a practical defense against DDoS attacks. It emerges as a strong candidate because it adapts dynamically to highly variable, adversarial traffic patterns where static or rule-based solutions often fail.

3.3.1 Comparing RL Approaches

Khozam et al. [19] present a Deep Reinforcement Learning–based mitigation system for SDN, where an intelligent agent adaptively tunes defense strategies against ICMP, TCP SYN, and UDP–based DDoS attacks. The authors favor RL over conventional ML pipelines because DDoS mitigation in SDN is a rapidly changing control problem: policies must adapt online to unseen conditions rather than merely classify traffic and apply static rules. In the Introduction, they argue that signature-, blockchain-, and rule-driven techniques are too inflexible for modern threats and can introduce unacceptable latency, whereas RL can discover effective tactics in dynamic, uncertain settings and refine them over time via feedback. The agent's state is built from flow-level traffic statistics, while its actions comprise fine-grained bandwidth allocation and traffic-control operations that the SDN controller enforces in the data plane, keeping decisions agile and responsive. A key design element is a performance-dependent reward that ties learning directly to

QoS metrics (latency/jitter of legitimate traffic), explicitly encoding the objective to mitigate attacks without degrading user experience. Overall, the approach aims to preserve network performance while mitigating live attacks by issuing adaptive, automated countermeasures aligned with the current network conditions.

Similarly, Simpson et al. [20] motivate reinforcement learning as an online defense for volume-based DDoS because network traffic is inherently non-stationary and labeled data are scarce or incomplete, which makes misuse-based or offline-supervised methods brittle; instead, RL "looks for (and controls) the effect rather than the cause," adapting under drift by learning from real-time consequences. The authors also highlight a long-standing "failure to launch" of ML-based anomaly detection in production: despite extensive research, deployments remain rare due to intrusion detection's extremely low tolerance for false positives, high error costs, limited open high-quality training data, and highly diverse, bursty traffic that shifts across timescales—hence the motivation for online control that learns from consequences rather than static signatures.

Concretely, they cast per-flow mitigation as sequential control and adopt a classical, low-latency RL stack, *Sarsa* with tile coding, trained strictly online (no replay, no neural nets). The state combines global link-load observations with per-flow measurements; for each source–destination pair, the agent selects the fraction of inbound traffic to drop, which is enforced in an SDN enviroment by installing a rule that drops each relevant packet with probability p. Per-flow decisions are emphasized as crucial for precise control and reduced collateral damage to legitimate traffic: unlike aggregate actions, per-flow policies can observe individual behavior (e.g., inter-arrival times, symmetry, response to drops) and apply tailored throttling, enabling forgiveness logic, performing better with congestion-aware protocols, and enhancing adaptability to dynamic traffic. In principle, the feedback-loop design allows continuous monitoring after an action to forgive mistakenly punished flows; if the state is tracked per flow and combined with the last applied action, by observing how a flow behaves after being penalized, the system can quickly separate mistaken punishments from genuinely malicious behavior.

Building on prior work, the authors investigated suitable features for feedback-loop control in online DDoS flow identification and identified two with particularly strong predictive power: *correspondence ratio* and Δ *send/receive rate*. The *correspondence ratio*: the ratio between a flow's outbound and inbound volumes for a source–destination pair helps expose asymmetric traffic patterns; values near zero indicate one–way, reply–poor exchanges that are characteristic of illegitimate volumetric activity. Finally, Δ *send/receive rate* measures how traffic rates change in response to the last action; this captures the behavioral reaction to bandwidth increases or reductions, providing a sensitive signal for online adaptation (benign and attack flows tend to respond differently to throttling or relief).

Ganesan et al. [21] investigate the practicality and effectiveness of using reinforcement learning to mitigate DoS attacks in SDN environments with programmable switches. The paper underscores SDN vulnerabilities, most notably the risk of overwhelming the centralized controller and saturating the control channel, and leverages the flexibility of P4-programmable switches together with RL's adaptiveness

to handle flooding-based attacks without consulting the controller for every malicious packet, while continuously learning to recognize new and previously unseen patterns.

The choice of RL is motivated by a key limitation of supervised ML: although models trained on labeled traffic can classify flows as benign or malicious with high accuracy, they adapt slowly to changing traffic conditions because they require (1) collecting new data, (2) labeling flows, (3) retraining, and (4) redeploying the model. Since this process is both time-consuming and often dependent on human intervention for labeling, its practicality in dynamic network environments is significantly reduced.

In practice, the authors integrate a PPO-based RL agent that learns attack signatures from flow features (e.g., packet counts and sizes) and translates its decisions into match–action rules directly installed in the P4 data plane to filter packets exhibiting those feature values. This closed-loop framework demonstrates that the agent can learn signatures that generalize beyond specific endpoints, i.e., independent of source/destination IPs, thereby remaining effective even against DDoS campaigns with spoofed sources.

Zhang et al. [22] propose an RL-driven throttling defense tailored to amplification DDoS, in which an agent learns filtering policies directly from live traffic rather than relying on static, port-based rules. Because amplified flows often resemble everyday traffic and appear to originate from commonly used servers, traditional packet-level filters struggle; by contrast, the authors argue that reinforcement learning is well-suited to mitigating such attacks. Concretely, they instantiate per-server agents (one per DNS amplifier) that, in fixed time windows, observe the two-dimensional state $\langle L_{\text{server}}, L_{\text{total}} \rangle$, the per-server load and the aggregate link load, choose an action (forward or drop), and receive rewards that promote delivery of legitimate traffic while penalizing congestion and overly dominant per-server load. The policy is learned via ϵ-greedy Q-learning, and the resulting decisions are implemented as a router-resident throttling strategy. Over time, the system adapts its filtering modes to varying traffic conditions, effectively discarding attack traffic while preserving normal flows.

3.3.2 Limitations and Open Issues

Taken together, these four studies validate the feasibility of RL-based DDoS mitigation, but they also expose cross-cutting fault lines that matter for real deployments. In moving from prototype to practice, recurring challenges surface: reliance on off-switch feature extraction and offline datasets, fragile reward design under non-stationary traffic, enforcement-only use of programmable data planes, and scalability pressures from state design and agent granularity. The following analysis distills these limitations and discusses their implications for deployability and safety.

External systems to extract flow-level features for the agent's state Several RL defenses construct the agent's state off-switch, requiring packet capture

and external feature extraction before batching features to the learner, which increases training latency and operational overhead. In Ganesan et al. [21], traffic is captured as pcaps and transformed into flow-level features using CICFlowMeter; these features feed a Gym wrapper that drives the RL agent, while the data plane only enforces learned rules and does not supply state to the agent directly. This architecture simplifies prototyping but requires packet captures and external processing during training, creating a tight coupling to control-plane telemetry. Similar CICFlowMeter-based pipelines are common across prior SDN RL defenses, incurring data-plane ⇔ control-plane transfer costs that add delay and load [19].

Dataset-based, offline training that may age poorly Some approaches train the agent offline by replaying labeled corpora (e.g., CICDDoS2019/CICIDS2017) and then evaluate actions against an "oracle" built from those labels [19],[21]. While this "offline RL" de-risks early exploration, it ties learning to the timeliness and representativeness of historical data, raising concerns about staleness as attack patterns evolve.

Using the programmable data plane only for enforcement, and focusing solely on flooding DDoS Among the surveyed works, Ganesan et al. [21] is the only defense that leverages a programmable data plane, but the switch is used primarily as an enforcement point: the RL agent runs off-switch and translates decisions into match-action rules pushed directly to the P4 pipeline. Moreover, the evaluation and framing explicitly focus on flooding-style DoS/DDoS, rather than a broader spectrum of attack classes.

Reward-function fragility: prior knowledge vs. proxy metrics Reward specification remains a central challenge. One line of work side-steps it with an "oracle" reward derived from labeled datasets, useful for feasibility studies but impractical online [21],[19]. Another defines reward via aggregate network metrics (e.g., penalizing congestion and rewarding the proportion of legitimate throughput, with additional priors), which may not faithfully capture per-flow maliciousness and can bias learning under mixed traffic [22],[20].

State Definition as a Bottleneck for Scalable RL Defenses A broader challenge across RL-for-networking is how to define the state: which traffic/telemetry features are truly informative, observable at line rate, and compact enough for learning. Zhang et al.'s [22] amplification defense illustrates the resource burden of instantiating a separate virtual agent per DNS amplifier and acknowledges that scaling to "hundreds of nodes" would demand deeper models and more complex state spaces. State design often requires substantial feature engineering (e.g., selecting packet/flow features that can both summarize behavior and be enforced as match–action keys), as shown in the [21] pipeline where the authors shortlist ports, flag counts, packet-length statistics, and per-direction rates to represent the environment and drive actions. Moreover, continuous-valued network measurements complicate tabular methods, to cope with continuous state spaces at line rate, some dataplane, oriented work adopts classic discretization schemes (tile coding)

with on-device Sarsa, highlighting a practical path to approximate continuous features within programmable hardware [20]. Crafting a state rich enough to capture dynamics yet light enough to run at scale underscores why state definition remains a challenge for RL-based defenses.

Table 3.1: Gap Analysis.

Design Aspect	**Khozam et al. [19]**	**Simpson et al. [20]**	**Ganesan et al. [21]**	**Zhang et al. [22]**	**Our Work**
Using the programmable data plane	✗	✗	✓	✗	✓
External systems to extract flow-level features for the agent's state	✓	✗	✓	✗	✗
Dataset-based, offline training	✓	✗	✓	✗	✗
Attack scope: amplification DDoS mitigation	✗	✓	✗	✓	✓
Reward-function: prior knowledge	✗	✓	✓	✓	✗
Reward-function: network metrics	✓	✓	✗	✓	✓

Chapter 4

Thesis Objectives

The previous chapters have highlighted how Distributed Denial of Service attacks have evolved into a persistent and large-scale threat to Internet availability, with amplification-based techniques standing out for their efficiency, scalability, and destructive potential. By exploiting protocol asymmetries and vulnerable third-party services, amplification attacks enable adversaries to generate massive traffic volumes with minimal effort, making traditional host-based and control-plane-centric defenses increasingly inadequate. As attack rates continue to reach terabit-per-second scales, mitigating malicious traffic as early as possible in the network has become a critical requirement rather than an optimization.

In parallel, recent advances in programmable data planes have sparked significant research interest in pushing security mechanisms directly into the forwarding path. By enabling line-rate packet processing, fine-grained telemetry, and stateful decision-making inside network devices, programmable switches offer a promising foundation for scalable, low-latency DDoS mitigation.

At the same time, the emergence of in-network machine learning has shown that data-plane programmability can be leveraged not only for static filtering, but also for adaptive and intelligent traffic control. However, when considering **RQ2**, despite this growing body of work, existing solutions largely rely on offline training, external control-plane support, or focus primarily on flooding-style attacks, leaving key challenges in amplification-aware, fully in-network learning unaddressed (**C2**).

To address **RQ1**, this work explores a research direction that has not been previously investigated to the best of our knowledge, namely the mitigation of amplification DDoS attacks entirely within the programmable data plane using online reinforcement learning. Specifically, we propose a Reinforcement Learning agent that learns directly from observed network traffic and discards malicious flows without relying on external systems or pre-existing datasets for training (**C1**).

In this way, the threat can be addressed at the network layer, avoiding the overhead of inspecting application-layer payloads. Moreover, the proposed agent can readily adapt to emerging malicious traffic patterns. To realize this vision, we design a learning agent that operates within the packet processing pipeline and relies exclusively on features that can be extracted at line rate.

To answer **RQ3**, the agent analyzes line-rate telemetry extracted from bidirectional flows between the victim and vulnerable public servers. The observed state

combines traffic asymmetry indicators derived from inbound and outbound flow statistics with a Quality-of-Service (QoS) indicator reflecting service degradation at the switch (**C3**). The integration of amplification-aware features with QoS feedback enables the agent to learn and enforce mitigation actions online, effectively filtering malicious traffic while safeguarding legitimate service performance.

Finally, to address **RQ4**, the proposed in-switch learning approach is experimentally compared against a static threshold-based filtering policy. The comparison aims to evaluate how adaptive learning decisions differ from fixed mitigation rules when facing varying traffic conditions. In particular, the study analyzes both the effectiveness of the mitigation strategy and the decision behavior of the system, highlighting how reinforcement learning can leverage temporal traffic context to perform more flexible and context-aware filtering decisions (**C4**).

Chapter 5

Methodology

5.1 System Modeling

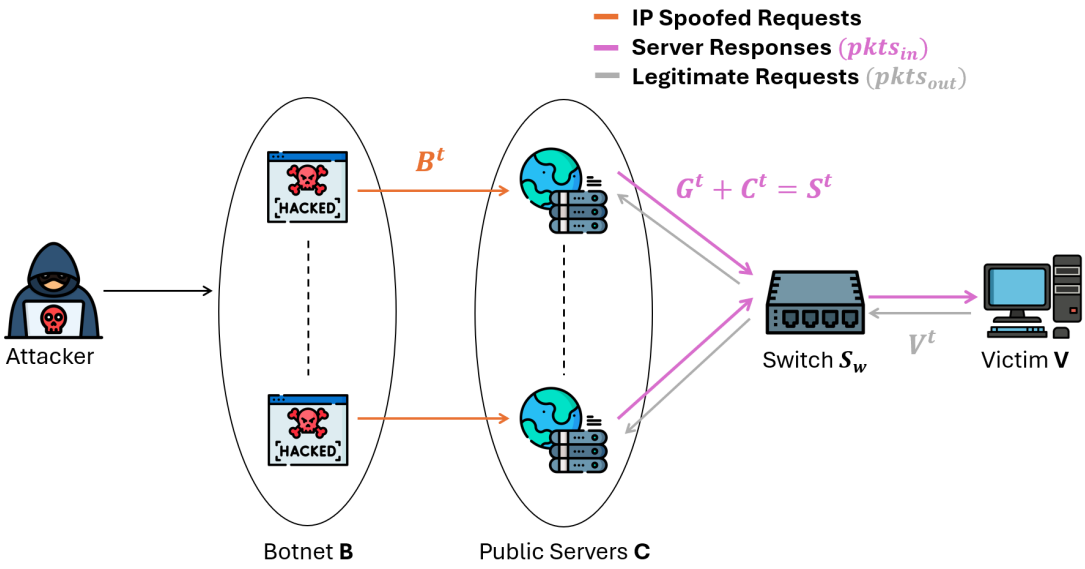

Figure 5.1: System model of an amplification-based DDoS attack scenario. A botnet B generates spoofed requests toward public servers C, which produce amplified responses directed to the victim V. Legitimate requests originating from the victim and amplified responses are forwarded through the programmable switch S_w.

Figure 5.1 illustrates the abstract system model considered in this work. We model a network scenario subject to an amplification-based DDoS attack, where an attacker controls a botnet that generates spoofed requests towards a set of public servers, which in turn produce amplified responses directed to a victim host.

Time is modeled as a discrete variable $t \in \mathbb{N}$, where each time step corresponds to a fixed observation window of duration Δt.

The system consists of the following abstract components:

- an attacker controlling a set of compromised hosts denoted as B (e.g., bots generating spoofed requests);
- a set of public servers exploited as reflectors denoted as C (e.g., DNS or NTP servers);
- a network switch forwarding traffic towards the victim S_{w} (a programmable data-plane switch);
- a victim host targeted by the attack denoted as V (the protected service endpoint).

During each time window t, the target generates a set of request packets that are forwarded by the switch toward one of the public servers. Let

$$\mathcal{V}^t = \{pkt_1^{out}, pkt_2^{out}, \ldots, pkt_{|\mathcal{V}^t|}^{out}\} \tag{5.1}$$

denotes the set of outbound packets observed during time window t, and $|\mathcal{V}^t|$ is the number of packets in the set.

As a consequence, the public servers generate response packets, forming the set

$$\mathcal{C}^t = \{pkt_1^{in}, pkt_2^{in}, \ldots, pkt_{|\mathcal{C}^t|}^{in}\}, \tag{5.2}$$

of outbound packets observed during time window t, and $|\mathcal{C}^t|$ is the number of packets in the set, which is forwarded by the switch toward the victim.

During each time window, the switch observes a set of flows, denoted as

$$\mathcal{F}_{\mathcal{C}} = \{f_1, f_2, \ldots, f_{\mathcal{C}}\}, \tag{5.3}$$

where each flow is identified by the 5-tuple

$$f_k = \langle IP_{src}, IP_{dst}, Protocol, Port_{src}, Port_{dst} \rangle. \tag{5.4}$$

While benign traffic may occasionally exhibit moderate request–response imbalance (e.g., due to DNSSEC or bulk replies), amplification attacks are characterized by persistent and significantly larger deviations that are sustained across multiple time windows. In contrast, under normal request–response operation, the Bandwidth Amplification Factor (AF_{b}) and the Packet Amplification Factor (AF_{p}), as defined in the theoretical background Section 2.1.3, between any request pkt^{in} and its corresponding response pkt^{out} associated with the same flow f_{k} satisfy

$$AF_{\mathrm{p}} = \frac{|\mathcal{C}^t|}{|\mathcal{V}^t|} \approx 1, \qquad AF_{\mathrm{b}} = \frac{\sum_{pkt^{in} \in \mathcal{C}^t} \mathrm{bytes}(pkt^{in})}{\sum_{pkt^{out} \in \mathcal{V}^t} \mathrm{bytes}(pkt^{out})} \approx 1. \tag{5.5}$$

If an attack occurs during a time window, the botnet generates a set of spoofed request packets directed toward the public servers, with the victim's IP address set as the destination address. Let

$$\mathcal{B}^t = \{pkt_1^{out}, pkt_2^{out}, \ldots, pkt_{|\mathcal{B}^t|}^{out}\} \tag{5.6}$$

denotes the set of spoofed requests observed during time window t, and $|\mathcal{B}^t|$ is the number of packets in the set.

For each spoofed request $pkt^{out} \in \mathcal{B}^t$, the public servers generate a set of amplified response packets, denoted by $\mathcal{G}(pkt^{out})$. The overall set of amplified response packets observed during the same time window is therefore given by

$$\mathcal{G}^t = \sum_{pkt^{out} \in \mathcal{B}^t} \mathcal{G}(pkt^{out}) = \{pkt_1^{in}, pkt_2^{in}, \ldots, pkt_{|\mathcal{G}^t|}^{in}\}, \tag{5.7}$$

that is the set of outbound packets observed during time window t, and $|\mathcal{G}^t|$ is the number of packets in the set, which is forwarded by the switch toward the victim. The resulting inbound traffic may saturate the victim's network and processing resources, leading to severe service degradation or denial of service.

In this case, the Packet Amplification Factor and the Bandwidth Amplification Factor satisfy

$$AF_{\mathrm{p}} = \frac{|\mathcal{G}^t|}{|\mathcal{B}^t|} \gg 1, \qquad AF_{\mathrm{b}} = \frac{\sum_{pkt^{in} \in \mathcal{G}^t} \mathrm{bytes}(pkt^{in})}{\sum_{pkt^{out} \in \mathcal{B}^t} \mathrm{bytes}(pkt^{out})} \approx 1. \tag{5.8}$$

As demonstrated in [20][22], the correspondence ratio between upstream and downstream traffic represents a highly informative feature for detecting malicious traffic. Traffic asymmetry constitutes a key behavioral indicator that helps distinguish legitimate traffic patterns, which typically exhibit balanced request-response exchanges, from illegitimate patterns generated during amplification attacks, where a limited volume of requests triggers a disproportionately large volume of responses.

Motivated by these observations, we introduce a generalized Asymmetry Index (AI) that jointly accounts for both packet-level and bandwidth-level imbalance to have a protocol-agnostic representation of the degree of asymmetry observed during a time window t. Specifically, the index is defined as a function of the Packet Amplification Factor AF_{p} and the Bandwidth Amplification Factor AF_{b},

$$\mathrm{AI}_t = f(AF_{\mathrm{b}}, AF_{\mathrm{p}})\,, \tag{5.9}$$

where the function $f(\cdot)$ is designed to map traffic asymmetries to a normalized range $\mathrm{AI}_t \in [0,1]$. Both factors are required, as amplification may arise either from a single request triggering a disproportionately large response, or from a single request eliciting multiple response packets, as observed across different amplification-prone protocols in Table 2.1.

Values of AI_t close to 0 indicate a strong asymmetry skewed toward the victim, where the outbound traffic from the victim to the servers (denoted as *out*) significantly exceeds the inbound traffic in the opposite direction (denoted as *in*). Conversely, values of AI_t close to 1 indicate asymmetry skewed toward the servers, where inbound traffic dominates outbound traffic. Values around 0.5 correspond to approximately symmetric traffic conditions, which are typically associated with benign request-response communication.

5.2 MDP Modeling

The objective of the mitigation system is to limit the impact of amplification-based DDoS attacks by reducing traffic asymmetry, while preserving legitimate request-response communication in order to avoid service degradation.

While programmable data planes enable fine-grained traffic monitoring and enforcement at line rate, relying on enforcing static rules in the data plane alone is insufficient to solve the mitigation problem formulated above. The ideal objective requires reasoning over multiple time windows, accounting for how mitigation actions influence both future traffic asymmetry and service degradation. However, programmable data-plane devices operate under strict constraints in terms of computation, memory, and supported operations, and have access only to local and instantaneous traffic observations. As a result, fixed thresholds and static rules cannot capture long-term dependencies and adapt to non-stationary traffic patterns. This limitation is particularly critical in the context of amplification-based DDoS attacks, which are particularly challenging to detect because their characteristics may evolve rapidly over time and the spoofed requests closely resemble benign traffic in terms of packet format, also the resulting responses appear as legitimate packets originating from public servers. As a consequence, packet-level filtering alone is often insufficient to capture the underlying malicious behavior. Instead, aggregate traffic characteristics, such as correspondence ratios, have been shown to be significantly more effective.

For these reasons, implementing the ideal mitigation strategy through deterministic data plane logic is impractical. The scenario instead exhibits an inherently sequential structure, where actions taken at time t affect subsequent traffic dynamics and service quality.Accordingly, we reformulate the mitigation task as a sequential decision-making problem, which can be naturally modeled as a Markov Decision Process (MDP), which provides a framework for modeling sequential decision-making under uncertainty [23]. An MDP is defined by the tuple

$$\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle, \tag{5.10}$$

where $\mathcal{S}$ denotes the state space, $\mathcal{A}$ the action space, P the state transition dynamics, R the reward function, and $\gamma \in (0,1)$ the discount factor.

Time is modeled as a sequence of discrete windows indexed by t. At each time window, the agent operates at the programmable data plane and observes a state $s_t \in \mathcal{S}$ that summarizes the current traffic conditions and indicators of service degradation observed at the switch. Based on the observed state, the agent selects a mitigation action $a_t \in \mathcal{A}$, which affects how traffic is handled during the current time window.

After applying action a_t, the system then transitions to a new state s_{t+1} according to a probabilistic transition function $P(s_{t+1} \mid s_t, a_t)$. The agent then receives a scalar reward $r_t = R(s_t, a_t)$, which reflects the effectiveness of the mitigation action in reducing attack impact while preserving service quality.

The discount factor γ captures the trade-off between immediate mitigation effects and long-term system performance, emphasizing that mitigation decisions taken at a given time window may have lasting consequences in the future.

However, the state transition dynamics are unknown and depend on external factors such as traffic fluctuations and adaptive attacker behavior, making explicit modeling impractical. As a result, classical solution methods for Markov Decision Processes are not applicable. The mitigation task is therefore addressed using a model-free reinforcement learning approach, in which the agent perceives possibly imperfect state information, selects actions that influence the evolution of the environment, and optimizes a long-term objective expressed through rewards, enabling adaptive mitigation policies that evolve over time while remaining compatible with the constraints of programmable data-plane environments.

5.3 Reinforcement Learning Formulation

In this work, we adopt a value-based reinforcement learning approach based on Q-learning, as it supports online learning with lightweight computations compatible with programmable data-plane constraints. A detailed motivation for this choice is provided in Section 5.4.

Since Q-learning operates on a state–action–reward formulation, in this section we describe the reinforcement learning formulation of the mitigation problem by instantiating the Markov Decision Process introduced above. In particular, the programmable switch is modeled as a learning agent that observes traffic statistics on a per-flow basis, selects mitigation actions for each flow, and receives feedback in the form of rewards. The following paragraphs define the traffic asymmetry metrics, state representation, action space, and reward function used by the agent to learn an effective mitigation policy.

Traffic Asymmetry Metric To quantify per-flow traffic imbalance, traffic volumes and packet counts are discretized into B bins, such that

$$\mathrm{bin}(\cdot) \in \{0, \ldots, B-1\}. \tag{5.11}$$

The bandwidth and packet amplification factors for flow f_k are computed as

$$AF_{\mathrm{b}}^{(k)} = h\big[\mathrm{bin}(\mathrm{bytes}(\mathcal{S}_k^t)) - \mathrm{bin}(\mathrm{bytes}(\mathcal{V}_k^t))\big], \qquad AF_{\mathrm{p}}^{(k)} = h\big[\mathrm{bin}(|\mathcal{S}_k^t|) - \mathrm{bin}(|\mathcal{V}_k^t|)\big], \tag{5.12}$$

where

$$h(x) = \frac{x}{B-1}. \tag{5.13}$$

and the function $\mathrm{bytes}(\cdot)$ returns the corresponding traffic volume measured in bytes and $\mathcal{S}_k^t$ and $\mathcal{V}_k^t$ denote the sets of inbound and outbound packets associated with flow f_k during time window t, respectively.

In particular, $\mathcal{S}_k^t$ is defined as

$$\mathcal{S}_k^t = \mathcal{C}_k^t \cup \mathcal{G}_k^t, \tag{5.14}$$

where $\mathcal{C}_k^t$ represents the legitimate response packets generated by the servers in reply to victim requests, and $\mathcal{G}_k^t$ denotes the amplified response packets triggered by spoofed attacker requests.

The discretization is motivated by the constraints of programmable data-plane architectures, which support only a limited set of arithmetic operations and typically do not allow general-purpose division. As a result, continuous-valued ratios are replaced by discrete representations that can be efficiently computed using match–action tables, where the output corresponding to a given pair of operands is precomputed and stored.

The resulting Asymmetry Index for flow f_k at time window t is defined as

$$\mathrm{AI}_t^{(k)} = 50 \cdot \left(1 + \left[w_\mathrm{b} AF_\mathrm{b}^{(k)} + w_\mathrm{p} AF_\mathrm{p}^{(k)}\right]\right), \tag{5.15}$$

where the weighting coefficients satisfy $w_\mathrm{b}+w_\mathrm{p} = 1$ and $0 \leq w_\mathrm{b}, w_\mathrm{p} \leq 1.$. Since both $AF_\mathrm{b}^{(k)}$ and $AF_\mathrm{p}^{(k)}$ take values in the interval $[-1,1]$, the weighted sum in brackets is also bounded in $[-1,1]$. The index is in the range [0,100] to ensure compatibility with programmable data-plane arithmetic. Values of $\mathrm{AI}_t^{(k)} \approx 50$ indicate symmetric traffic conditions, values close to 0 indicate victim-side traffic imbalance, and values close to 100 indicate server-side traffic imbalance.

State Representation Time is divided into discrete windows indexed by t. At the end of each time window, the agent observes an aggregated traffic asymmetry indicator and a service degradation signal resulting from the previously applied mitigation decisions.

The state at time t is defined as:

$$s_t = \big(\mathbf{AI}_{t-1}, \mathbf{a}_{t-1}, \mathrm{ServDegr}_t\big), \tag{5.16}$$

where:

- $\mathbf{AI}_{t-1} = (AI_{t-1}, AI_{t-2}, \ldots, AI_{t-H})$ is a finite history vector of past asymmetry index values, computed over the previous H time windows;
- $\mathbf{a}_{t-1} = (a_{t-1}, a_{t-2}, \ldots, a_{t-H})$ is the corresponding history of mitigation actions applied during those windows;
- $\mathrm{ServDegr}_t$ encodes whether mitigation decisions have negatively impacted legitimate traffic, due to excessive packet drops over consecutive windows.

The state representation is extended to include, in addition to the current traffic indicators (AI_{t-1}, a_{t-1}), both the asymmetry index and the action applied in the previous time windows. This design choice is motivated by two distinct considerations.

First, the asymmetry index does not have an absolute semantic meaning when interpreted in isolation. The same asymmetry value may correspond to benign traffic conditions, ongoing attack activity, or policy-induced distortion depending on whether traffic was previously allowed or dropped. For instance, a low asymmetry level observed under a permissive policy may indicate normal traffic fluctuations, whereas the same value observed under an aggressive dropping policy may reflect

over-mitigation effects. By associating each asymmetry value with the action under which it was generated, the agent can correctly interpret the traffic condition and distinguish genuine imbalance introduced by its own policy.

Second, incorporating the previous asymmetry levels and the corresponding actions is necessary to ensure correct reward assignment. If the reward was computed solely based on the current asymmetry index and action, without considering the state in which the decision was made, an abrupt traffic change, such as the sudden onset of an attack, could cause the agent to receive negative rewards even when its action was fully consistent with the observed traffic pattern.

In other words, the agent was implicitly held responsible for exogenous dynamics beyond its control, leading to unstable and potentially misleading learning behavior. By explicitly encoding both the traffic metrics and the actions that produced them, the proposed formulation evaluates decisions in context, thereby preventing unjustified penalization and improving the stability and causal coherence of the learning process.

The history length H is treated as a design parameter and specified at implementation time. By explicitly augmenting the state with this bounded memory, the Markov property is preserved, as future transitions and rewards depend only on the current augmented state and action.

Including finite histories of both the asymmetry index and the applied actions allows the agent to model short-term temporal dependencies between mitigation decisions and their delayed effects on traffic dynamics and service quality. This approach is commonly adopted to partially mitigate limited observability at the data plane and has also been used in related works [20].

Action Space Mitigation decisions are applied independently on a per-flow basis. For each flow f_k, the action space is defined as

$$\mathcal{A}^t = \{\text{DROP}, \text{ALLOW}\}, \tag{5.17}$$

where the selected action determines whether packets belonging to flow f_k are forwarded or filtered during the current time window. In particular, when the action DROP is selected, packets associated with flow f_k are dropped in both directions, that is for traffic flowing from the victim to the servers and from the servers to the victim.

Reward Function The reward function is designed to drive the agent toward mitigation strategies that suppress attack-induced traffic asymmetry while avoiding unnecessary service disruption. The learning objective is therefore to promptly react to malicious imbalance and to prevent excessive degradation of legitimate traffic. At each time window t, the switch receives a scalar reward

$$r_t \in \{-1, 0, +1\}, \tag{5.18}$$

computed as the combination of an action-conditioned asymmetry component r_t^{AI} and a service degradation constraint.

Action-Conditioned Asymmetry Component. The current asymmetry index AI_{t-1} is not evaluated in isolation. Its interpretation depends on the action applied in that time window a_{t-1}, since the same asymmetry value may correspond to balanced traffic, insufficient mitigation, or excessive dropping depending on whether traffic was previously allowed or dropped.

When evaluating the action selected at time window $t-1$, the reward mechanism therefore considers the recent decisions and traffic history that led to the current traffic condition. In principle, the reward function could incorporate a longer temporal dependency and inspect multiple past decisions in order to more accurately attribute causality. However, for the sake of methodological clarity and to demonstrate feasibility in this implementation, the historical depth is limited to two decision windows ($H = 2$). This limited look-back is sufficient to capture corrective or persistent behaviors while maintaining a simple and interpretable formulation.

Accordingly, the asymmetry-based reward component r_t^{AI} is defined as a function of the pair (AI_{t-1}, a_{t-1}), optionally informed by the immediately preceding asymmetry observation AI_{t-2}.

Let β_L, β_M^-, β_M^+, and β_H denote symbolic thresholds defining low, moderate, and high asymmetry regions, with

$$\beta_L < \beta_M^- < \beta_M^+ < \beta_H \tag{5.19}$$

The asymmetry-based reward is defined as

$$r_t^{AI} = \begin{cases} 0, & \text{if } AI_{t-1} \in (\beta_M^-, \beta_M^+) \text{ and } a_{t-1} = \text{ALLOW and } AI_{t-2} \geq \beta_H, \\ +1, & \text{if } AI_{t-1} \in (\beta_M^-, \beta_M^+) \text{ and } a_{t-1} = \text{ALLOW}, \\ 0, & \text{if } AI_{t-1} \leq \beta_L \text{ and } a_{t-1} = \text{DROP and } AI_{t-2} \geq \beta_H, \\ -1, & \text{if } AI_{t-1} \leq \beta_L \text{ and } a_{t-1} = \text{DROP}, \\ 0, & \text{if } AI_{t-1} \geq \beta_H \text{ and } a_{t-1} = \text{ALLOW and } AI_{t-2} \leq \beta_M^+, \\ -1, & \text{if } AI_{t-1} \geq \beta_H \text{ and } a_{t-1} = \text{ALLOW}, \\ 0, & \text{otherwise.} \end{cases} \tag{5.20}$$

Considering the first pair of conditions in Eq. (5.20), the agent is rewarded when the observed asymmetry and the previously applied action are consistent with **balanced** traffic conditions. In particular, when the asymmetry lies in the moderate region and the previous action was `ALLOW`, the traffic is interpreted as symmetric and therefore benign. In this case, the agent receives a positive reward. However, if this apparent symmetry follows a decision taken in a prior window where the asymmetry was clearly in the high region, the reward is set to zero. This prevents the agent from being rewarded from allowing traffic during an ongoing attack phase.

The second pair of conditions in Eq. (5.20) describes the **overreaction** case. The agent is penalized when a dropping decision leads to traffic that is strongly asymmetric toward the victim side, i.e., when the asymmetry falls in the low region under a `DROP` policy. Such a condition indicates that no attack is likely in progress and that legitimate client–server communication is being unnecessarily suppressed.

Nevertheless, if the dropping decision follows a previous window characterized by high asymmetry, the reward is set to zero rather than negative. This avoids penalizing defensive actions that were reasonable given earlier evidence of potential attack activity.

Finally, the third pair of conditions in Eq. (5.20) captures the **underreaction** case. The agent is penalized when allowing traffic results in strong asymmetry toward the server side, which indicates insufficient mitigation during a likely attack condition. However, if the permissive decision was taken when the asymmetry two windows earlier was balanced or only mildly asymmetric toward the victim side, the reward is again neutral. This ensures that the agent is not punished for decisions that were coherent with the information available at the time they were made.

Overall, this formulation encourages corrective behavior while preventing incorrect credit assignment due to abrupt traffic dynamics. The agent is evaluated based on the consistency of its decisions with the recent traffic evolution, rather than solely on the absolute asymmetry observed in the current window.

Service Degradation Constraint. To balance security and service availability, the final reward incorporates the service degradation counter $ServDegr_t$.

Service degradation is modeled as a cumulative counter that increases upon drop decisions and decreases when traffic is forwarded.

If the cumulative degradation exceeds a predefined threshold X, the reward is adjusted as follows:

$$r_t = \begin{cases} -1, & \text{if } ServDegr_t > X \text{ and } r_t^{AI} \neq +1, \\ 0, & \text{if } ServDegr_t > X \text{ and } r_t^{AI} = +1, \\ r_t^{AI}, & \text{otherwise.} \end{cases} \tag{5.21}$$

This formulation reflects the practical observation that repeatedly dropping traffic may severely degrade service quality. For example, in the case of DNS traffic, persistent packet drops can prevent legitimate clients from receiving responses, leading to failed name resolution and poor user experience. In contrast, allowing traffic in ambiguous conditions may help preserve service continuity.

An important assumption underlying this design is that amplification-based attacks do not persist longer than N consecutive time windows. Therefore, the service degradation threshold X is chosen such that $X \leq N$, thus this design ensures that the service degradation mechanism regulates unnecessary aggressiveness but does not interfere with legitimate defensive behavior during attack phases.

5.4 Reinforcement Learning Algorithm

Given the reinforcement learning formulation described in the previous section, a suitable learning algorithm must satisfy several practical and architectural constraints imposed by the mitigation problem and by programmable data-plane environments.

First, the state transition dynamics are unknown and depend on external traffic fluctuations as well as adaptive attacker behavior. This precludes the use of model-based approaches and motivates the adoption of a model-free reinforcement learning algorithm capable of learning directly from interaction with the environment.

Second, the state and action spaces defined in Section 5.3 are discrete and relatively compact, as traffic statistics are discretized through binning and mitigation actions are binary. Under these conditions, tabular reinforcement learning methods are well suited, as they allow explicit representation of state-action values without requiring function approximation.

Third, programmable data planes impose strict constraints in terms of computational complexity, memory usage, and supported operations. Learning algorithms deployed in this context must rely on simple arithmetic operations, bounded memory, and incremental updates.

Q-learning satisfies these requirements, as demonstrated by recent works [17], which shows that the Q-learning update rule and state representation can be efficiently mapped onto data-plane primitives, such as registers and match-action tables, enabling fully in-network reinforcement learning without reliance on external controllers or offline training. QCMP further demonstrates that both inference and online training can be performed at line rate with negligible hardware resource consumption, validating the practicality of deploying learning agents directly within programmable switches.

Finally, Q-learning naturally supports per-flow learning, enabling the switch to maintain independent state-action value estimates for different flows. This aligns with the per-flow mitigation model adopted in this work and allows the learning process to capture heterogeneous traffic behaviors across servers.

For these reasons, Q-learning is adopted as the reinforcement learning algorithm to learn adaptive mitigation policies directly within the programmable data plane.

As explained in the Section 3.2.1, Q-learning learns an action-value function $Q(s,a)$, which estimates the expected cumulative discounted reward obtained by selecting action a in state s and following the learned policy thereafter.

For each flow f_i, the Q-function is updated independently, at time window t, after observing state $s_t^{(i)}$, selecting action $a_t^{(i)}$, and receiving reward $r_{t-1}^{(i)}$, the Q-value is updated according to

$$Q_{t-1}\left(s_{t-1}^{(i)}, a_{t-1}^{(i)}\right) = Q_{t-1}\left(s_{t-1}^{(i)}, a_{t-1}^{(i)}\right) + \alpha\left[r_t^{(i)} + \gamma \max_{a\in\mathcal{A}} Q_t\left(s_t^{(i)}, a\right) - Q_{t-1}\left(s_{t-1}^{(i)}, a_{t-1}^{(i)}\right)\right], \tag{5.22}$$

where $\alpha \in (0,1]$ is the learning rate and $\gamma \in (0,1)$ is the discount factor.

Algorithm 1 Time-Window-Based Q-Learning for Traffic Throttling

Initialize Q(s,a) arbitrarily
Set learning rate α, discount factor γ, exploration rate ϵ
Define time window duration T
for each episode **do**
    Initialize environment
    Observe initial state s_0
    while episode not terminated **do**
        // Action selection (per time window)
        Select action a_t from s_t using ϵ-greedy policy
        // Apply action during window T
        Execute a_t for the entire time window T
        // Observe next state and reward
        Measure traffic statistics over T
        Compute next state s_{t+1}
        Compute reward r_t
        // Q-learning update
        $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$
        $s_t \leftarrow s_{t+1}$
    end while
end for

Chapter 6

Implementation

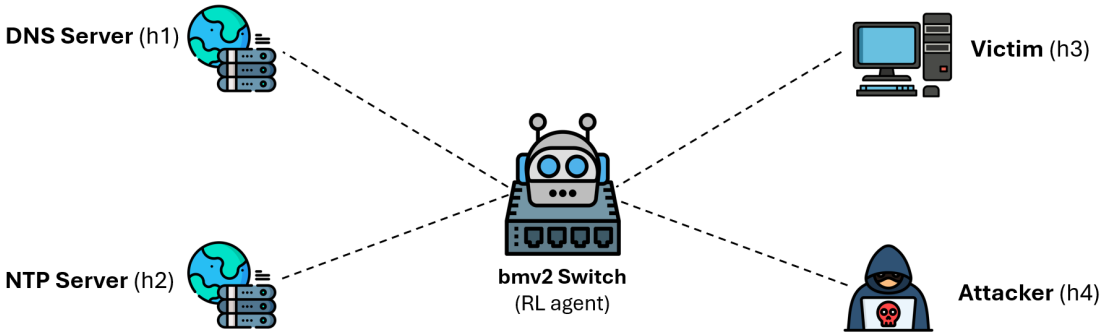

Figure 6.1: Network topology used in the implementation. A bmv2 switch operates as the RL agent and forwards traffic between the victim, the attacker, and DNS and NTP servers are exploited as reflectors in an amplification-based DDoS scenario.

In this section, we present the implementation of our in-switch RL-based solution for the mitigation of amplification DDoS attacks. The proposed system is designed to operate directly in the data plane, enabling fast and scalable attack mitigation without relying on per-packet decisions from the control plane. Based on the network topology introduced in Figure 5.1 in the previous section, we consider a simplified yet representative scenario as depicted in Figure 6.1.

For clarity and reproducibility, the following assumptions are adopted in the implementation:

- **History length.** The agent state includes traffic statistics from the last H time windows, where $H = 2$.
- **Number of legitimate servers.** The topology includes $|C| = 2$ legitimate servers generating traffic.
- **Number of attackers.** A single logical attacker entity is considered, modeling spoofed amplification traffic without explicitly simulating a distributed botnet.

- **Time window duration.** Traffic decisions are taken at fixed intervals of duration $T = 10$.
- **Service degradation model.** Service degradation is modeled as a cumulative counter that increases by one unit for each `DROP` decision and decreases by one unit for each `ALLOW` decision. Consistently with what is discussed in Section 5.3, the degradation threshold X is set to $X = 8$. Furthermore, we assume that amplification-based attacks do not persist longer than $N = 8$ consecutive time windows.
- **Asymmetry thresholds.** The asymmetry thresholds are set, respectively, to $\beta_L = 0.35$, $\beta_M^- = 0.45$, $\beta_M^+ = 0.55$, and $\beta_H = 0.65$.

6.1 P4 Data Plane

The data plane is implemented using the Behavioral Model v2 (BMv2) framework, which provides a software-based reference implementation of P4-programmable switches. BMv2 is designed to accurately emulate the behavior of programmable data planes, enabling development, testing, and validation of P4 programs in a controlled and reproducible environment.

As execution target, we adopt the *simple_switch* implementation, which is the most widely used BMv2 target and represents the de-facto standard for P4 experimentation [2][24]. The *simple_switch* target implements a complete packet processing pipeline and supports match-action tables, counters, meters, and registers, making it particularly suitable for prototyping advanced in-network functionalities.

The switch follows the *v1model* architecture, which defines a standardized P4 pipeline composed of **Parser**, **Ingress** and **Egress** match-action stages, and a **Deparser**. This combination provides a stable, hardware-agnostic, and well-established platform for evaluating in-switch reinforcement learning mechanisms.

6.1.1 Header and Parser

The P4 program begins with a set of constant definitions that provide a shared configuration for packet classification, flow management, and reinforcement learning logic implemented in the data plane.

These constants serve three main purposes. First, they define protocol identifiers (e.g., EtherType and transport-layer protocol numbers) required for correctly parsing and classifying incoming packets. Second, they introduce timing and discretization parameters used to aggregate traffic statistics over fixed observation windows and to encode continuous features into discrete bins suitable for table indexing. Finally, they specify RL-related parameters, including action identifiers, exploration thresholds, and Q-table dimensions, which enable a lightweight in-switch implementation of the algorithm.

Additionally, the constant definitions include parameters used to encode signed Q-values within the unsigned bit-level representation supported by the P4 data

plane. Since P4 registers and arithmetic operations are primarily defined over unsigned bit fields, Q-values are represented using a fixed-width signed range mapped onto an unsigned domain. The constants Q_OFFSET, Q_MIN, and Q_MAX define, respectively, the offset applied to shift signed values into the unsigned range and the minimum and maximum representable Q-values. This encoding enables compact storage and comparison of reinforcement learning values directly within switch registers while preserving their signed semantics.

By centralizing these values as compile-time constants, the design improves readability and portability, while allowing the behavior of the agent to be tuned without modifying the core packet-processing logic.

The packet processing pipeline relies on explicit header and metadata definitions to expose protocol fields and maintain state across stages, as shown in Listing 6.1. Standard protocol headers are defined for Ethernet, IPv4, and UDP packets, enabling the parser to extract all relevant network-layer and transport-layer fields (lines 4-30). In particular, the UDP header exposes source and destination ports, which are essential for identifying amplification services.

In addition to protocol headers, a dedicated metadata structure is defined to store per-flow state and reinforcement learning variables (lines 32-52). The metadata includes a unique flow identifier and direction flags to distinguish incoming and outgoing traffic, along with per-flow packet and byte counters used to quantify traffic asymmetry. These counters enable the computation of amplification factors in terms of both packet count and byte volume in the Egress block.

To support in-switch reinforcement learning, the metadata struct also keeps discretized feature values, encoded as fixed-width bins, which are used to index the Q-table and represent the current state of the environment (lines 41-46). Additional fields are used to trigger agent execution, store the last action taken, and maintain a temporal reward signal derived from traffic asymmetry metrics.

```
1  typedef bit<48> macAddr_t;
2  typedef bit<32> ip4Addr_t;
3
4  header ethernet_t {
5      macAddr_t dstAddr;
6      macAddr_t srcAddr;
7      bit<16> etherType;
8  }
9
10 header ipv4_t {
11     bit<4> version;
12     bit<4> ihl;
13     bit<8> diffserv;
14     bit<16> totalLen;
15     bit<16> identification;
16     bit<3> flags;
17     bit<13> fragOffset;
18     bit<8> ttl;
19     bit<8> protocol;
20     bit<16> hdrChecksum;
```

```
    ip4Addr_t srcAddr;
    ip4Addr_t dstAddr;
}

header udp_t {
    bit<16> srcPort;
    bit<16> dstPort;
    bit<16> len;
    bit<16> checksum;
}

struct metadata {
    bit<32> flow_id; // Unique per-flow identifier
    bit<16> port_to_check; // UDP port to inspect for amplification
    bit<1> direction; // INCOMING or OUTGOING
    bit<1> is_amplifiable_service;
    bit<PKT_COUNT_BITS> in_pkt_count; // Packet count incoming
    bit<PKT_COUNT_BITS> out_pkt_count; // Packet count outgoing
    bit<BYTE_COUNT_BITS> in_byte_count; // Byte count incoming
    bit<BYTE_COUNT_BITS> out_byte_count; // Byte count outgoing
    bit<3> bin; // Discretized feature index
    bit<1> agent_trigger; // Flag to activate RL decision
    bit<3> f0;
    bit<3> f1; // Binned features
    bit<3> f2; // for RL-related computations
    bit<3> f3;
    bit<16> amplification_factor_byte;
    bit<16> amplification_factor_pkt;
    bit<8> asymmetry_index; // Asymmetry index for RL reward
    int<8> reward; // Temporal reward signal
    bit<1> last_action_taken; // Last RL action: ALLOW or DROP
    bit<4> service_degradation;
    bit<1> prev_action;
    bit<5> prev_asymmetry_index_bin;
    bit<1> curr_action;
    bit<5> curr_asymmetry_index_bin;
};

struct headers {
    ethernet_t ethernet;
    ipv4_t ipv4;
    udp_t udp;
}
```

Listing 6.1: Header type definitions and metadata structures for per-flow and RL state

The parser in Listing 6.2 implements the logic required to extract standard protocol headers and to drive the construction of the corresponding parse graph. Each packet parsing process starts from a common start state and proceeds through a sequence of parsing states based on the values of previously extracted header fields.

The parser extracts the Ethernet header and uses the EtherType field to identify IPv4 packets, while non-IPv4 traffic is accepted without further parsing. For IPv4 packets, the IPv4 header is extracted and the protocol field is inspected to selectively parse UDP headers. All other packets are directly accepted.

In this work, the resulting parse graph is simple, as the system relies exclusively on standard protocol headers and does not introduce any custom encapsulation formats.

```
parser MyParser(packet_in packet,
                out headers hdr,
                inout metadata meta,
                inout standard_metadata_t standard_metadata) {

    state start {
        transition parse_ethernet;
    }

    state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            TYPE_IPV4: parse_ipv4;
            default: accept;
        }
    }

    state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition select(hdr.ipv4.protocol){
            TYPE_UDP: parse_udp;
            default: accept;
        }
    }

    state parse_udp {
        packet.extract(hdr.udp);
        transition accept;
    }
}
```

Listing 6.2: Parser implementation for Ethernet, IPv4, and UDP packets

6.1.2 Ingress

The ingress control block begins by declaring a set of temporary variables and persistent registers used to maintain per-flow traffic state across packets. Temporary variables are defined to support intermediate computations during packet processing without allocating persistent memory.

Separate registers are maintained for incoming and outgoing packet (*in_pkt_count_reg* and *out_pkt_count_reg*), and byte counts (*in_byte_count_reg* and *out_byte_count_reg*),

enabling the switch to capture traffic asymmetry at the flow level. In addition, a timestamp register records the arrival time of the first packet within each monitoring window, supporting time-based aggregation of per-flow traffic statistics.

The ingress stage in Listing 6.3 defines a set of match-action tables and actions that collectively implement packet forwarding, flow direction identification, and amplification service detection.

The *ipv4_lpm* table (line 12) is responsible for layer-3 forwarding decisions based on longest-prefix matching on the IPv4 destination address. When a matching entry is found, the *ipv4_forward* action updates the Ethernet source and destination addresses, decrements the IPv4 time-to-live field, and assigns the appropriate egress port. Packets that do not match any forwarding rule are handled by the drop action, which marks them for immediate discard.

Following routing, the *packet_direction* table (line 37) classifies packets as incoming or outgoing traffic and assigns a bounded per-flow identifier. Depending on the matched prefix, the table invokes either the *get_flow_id_in* or *get_flow_id_out* action. These actions set the traffic direction and compute a flow identifier using a CRC32 hash over a selected subset of packet header fields.

The *check_amplification_service* table (line 58) determines whether a packet is associated with a known amplification service by performing an exact match on the inspected transport-layer port. The *amplifiable_service* and *not_amplifiable_service* actions set a metadata flag that is later used to trigger feature extraction and reinforcement learning decisions.

Finally, the *read_and_clear* action (line 71) populates the metadata with packet and byte counts accumulated over the monitoring window in the ingress registers, making them available to the egress block, and simultaneously clears the corresponding register entries.

```
1  control MyIngress(inout headers hdr,
2                    inout metadata meta,
3                    inout standard_metadata_t standard_metadata) {
4
5      action drop() {
6          mark_to_drop(standard_metadata);
7      }
8
9      action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
10         standard_metadata.egress_spec = port;
11         hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
12         hdr.ethernet.dstAddr = dstAddr;
13         hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
14     }
15
16     table ipv4_lpm {
17         key = {
18             hdr.ipv4.dstAddr: lpm;
19         }
20         actions = {
21             ipv4_forward;
```

```
22              drop;
23          }
24          size = 1024;
25          default_action = drop();
26      }
27
28      action get_flow_id_in() {
29          meta.direction = INCOMING;
30          hash(meta.flow_id, HashAlgorithm.crc32, (bit<32>)0,
31               {hdr.ipv4.srcAddr, hdr.ipv4.dstAddr, hdr.ipv4.protocol,
                     hdr.udp.srcPort, hdr.udp.dstPort},
32               (bit<32>)MAX_FLOWS);
33      }
34      action get_flow_id_out() {
35          meta.direction = OUTGOING;
36          hash(meta.flow_id, HashAlgorithm.crc32, (bit<32>)0,
37               {hdr.ipv4.dstAddr, hdr.ipv4.srcAddr, hdr.ipv4.protocol,
                     hdr.udp.dstPort, hdr.udp.srcPort},
38               (bit<32>)MAX_FLOWS);
39      }
40
41      table packet_direction {
42          key = {
43              hdr.ipv4.dstAddr: lpm;
44          }
45          actions = {
46              get_flow_id_in;
47              get_flow_id_out;
48              drop;
49          }
50          size = 4;
51          default_action = drop();
52      }
53
54      action amplifiable_service() {
55          meta.is_amplifiable_service = 1;
56      }
57
58      action not_amplifiable_service() {
59          meta.is_amplifiable_service = 0;
60      }
61
62      table check_amplification_service {
63          key = {
64              meta.port_to_check: exact;
65          }
66          actions = {
67              amplifiable_service;
68              not_amplifiable_service;
69          }
70          size = 1024;
```

```
71          default_action = not_amplifiable_service();
72      }
73
74      // Single action to read all registers at index 'idx', populate
            the metadata, then clear the registers back to 0.
75      action read_and_clear(bit<32> idx) {
76          in_pkt_count_reg.read(meta.in_pkt_count, idx);
77          in_pkt_count_reg.write(idx, (bit<16>)0);
78
79          out_pkt_count_reg.read(meta.out_pkt_count, idx);
80          out_pkt_count_reg.write(idx, (bit<16>)0);
81
82          in_byte_count_reg.read(meta.in_byte_count, idx);
83          in_byte_count_reg.write(idx, (bit<32>)0);
84
85          out_byte_count_reg.read(meta.out_byte_count, idx);
86          out_byte_count_reg.write(idx, (bit<32>)0);
87      }
```

Listing 6.3: Match-action tables for routing, direction identification, and service detection

The Ingress apply block in Listing 6.4 orchestrates the execution of the ingress pipeline by conditionally invoking match-action tables and updating per-flow state. Processing is restricted to IPv4 packets, for which routing is first performed through the *ipv4_lpm* table. Subsequent logic is applied only to UDP traffic, reflecting the focus on amplification-based attacks. Packets arriving from the attacker-facing port, which correspond to spoofed request traffic generated by the attacker, are explicitly excluded from further processing to avoid unnecessary state updates.

For eligible packets, the ingress pipeline determines traffic direction and assigns a flow identifier through the *packet_direction* table. Based on the detected direction, the relevant UDP port is selected and stored in metadata to identify the potential service associated with the packet. The *check_amplification_service* table is then applied to determine whether the packet belongs to a known amplifiable service, enabling selective activation of monitoring logic.

When an amplifiable service is detected, the ingress block enforces a time-window–based aggregation mechanism. For each flow, the timestamp of the first packet in the current window is retrieved and compared against the global ingress timestamp. If the monitoring window has elapsed, accumulated per-flow counters are read and cleared, a new window is initialized, and a trigger is set to signal the reinforcement learning agent to select a new action for the flow. Otherwise, the current window is maintained and thus packet and byte counters are updated according to the packet direction.

The ingress stage is responsible for window-based flow monitoring and for providing the information required by the agent to make mitigation decisions in subsequent processing stages.

```
1   apply {
2       if (hdr.ipv4.isValid()) {
```

```
        ipv4_lpm.apply();
        if (hdr.udp.isValid()) {
            // If packet comes from the attacker, avoid any processing and just forward
            if (standard_metadata.ingress_port != PORT_TO_ATTACKER) {
                // Determine port to check based on packet direction
                packet_direction.apply();

                // If packet is incoming (from network -> our host), look at the UDP source port, instead if it is
                // outgoing (our host -> network), look at the UDP destination port
                if (meta.direction == INCOMING) {
                    meta.port_to_check = hdr.udp.srcPort;
                } else {
                    meta.port_to_check = hdr.udp.dstPort;
                }

                // Detect amplifiable service
                check_amplification_service.apply();

                if(meta.is_amplifiable_service == 1){
                    // Read the stored timestamp for this flow (first packet in the current window)
                    first_ts.read(tmp48, meta.flow_id);

                    // If we already have a timestamp (i.e., not zero)...
                    if (tmp48 > 0) {
                        // If the elapsed time since the first packet exceeds the time window...
                        if (standard_metadata.ingress_global_timestamp - tmp48 > time_window) {
                            // Trigger read_and_clear to process the accumulated flow state
                            read_and_clear(meta.flow_id);

                            // Start a new time window by recording this packet's timestamp
                            first_ts.write(meta.flow_id, standard_metadata.ingress_global_timestamp);

                            // Signal that the agent should select a new action for this flow
                            meta.agent_trigger = ACTIVE;
                        }
                    }
                    else{
                        // Otherwise (no timestamp yet), record the current packet's timestamp
                        // as the first packet in this time window
```

```
43                      first_ts.write(meta.flow_id,
                            standard_metadata.ingress_global_timestamp);
44                      }
45
46                      // Update packet/byte counters per direction
47                      if(meta.direction == INCOMING ){
48                          in_pkt_count_reg.read(tmp16, meta.flow_id);
49                          tmp16 = tmp16 + 1;
50                          in_pkt_count_reg.write(meta.flow_id, tmp16);
51
52                          in_byte_count_reg.read(tmp32, meta.flow_id);
53                          tmp32 = tmp32 + standard_metadata.packet_length;
54                          in_byte_count_reg.write(meta.flow_id, tmp32);
55                      }
56                      else{ // meta.direction == OUTGOING
57                          out_pkt_count_reg.read(tmp16, meta.flow_id);
58                          tmp16 = tmp16 + 1;
59                          out_pkt_count_reg.write(meta.flow_id, tmp16);
60
61                          out_byte_count_reg.read(tmp32, meta.flow_id);
62                          tmp32 = tmp32 + standard_metadata.packet_length;
63                          out_byte_count_reg.write(meta.flow_id, tmp32);
64                      }
65                  }
66              }
67          }
68      }
69  }
```

Listing 6.4: Ingress apply logic for flow monitoring and window-based state update

6.1.3 Egress

The egress control block represents the core of the proposed in-switch reinforcement learning implementation. While the ingress pipeline is responsible for traffic monitoring and state aggregation, the egress block performs the actual learning, decision-making, and policy update operations. All reinforcement learning state is maintained and updated in this stage, enabling the switch to autonomously adapt its mitigation behavior based on observed traffic patterns.

The egress block defines a set of registers that collectively implement the Q-learning model. A packed Q-table stores the action-value estimates for each discretized state, while additional per-flow registers track the previously visited state, the last actions taken, the cumulative service degradation counter, the previous asymmetry index, and the number of elapsed time windows. Global registers are also used to support ϵ-greedy exploration through random number generation and to distinguish the initial learning step from subsequent updates.

Local variables declared in this block are used to store intermediate results of

the Q-learning computations, such as the estimated future reward, the temporal-difference error, and the updated action-value. The detailed learning equations and their step-by-step implementation are discussed later in this section.

The Listing 6.5 defines a sequence of match-action tables used to discretize traffic statistics, derive amplification-related metrics, and compute the reward signal required by the reinforcement learning algorithm.

The *in_pkt_count_bin*, *out_pkt_count_bin*, *in_byte_count_bin*, and *out_byte_count_bin* tables perform range-based matching on per-flow packet and byte counters and invoke the *set_bin* to assign the corresponding discretized bin value to the metadata. These tables convert continuous traffic measurements into a compact, finite representation suitable for Q-table indexing.

Based on the discretized features, the *amplification_factor_byte* and *amplification_factor_pkt* tables compute amplification factors in terms of byte volume and packet count, respectively. Each table matches on pairs of binned features and applies the corresponding action to store the derived amplification factor in metadata (*set_amplification_factor_byte* or *set_amplification_factor_pkt*). This separation allows the switch to independently model amplification effects in both dimensions.

The *asymmetry_index* table combines the computed amplification factors with the inspected service port to derive a single asymmetry indicator. Including the service port in the matching key enables service-specific weighting of the amplification factors, allowing different services to influence the asymmetry computation according to configurable weights. Through the *set_asymmetry_index* action, the table encodes the degree of traffic imbalance observed for a given flow.

The reward computation is implemented through a dedicated match-action table that evaluates the action taken in the previous time window in conjunction with the recent asymmetry history and the service degradation state.

Rather than matching on the continuous asymmetry index, the switch quantizes it into small fixed-width bins (e.g., intervals of a few units). This discretization is not intended to simplify the traffic model, but rather to ensure that the reward table remains tractable in size and feasible within data plane constraints. Using fine-grained bins preserves the granularity of the asymmetry signal and consistency with the theoretical formulation while avoiding an excessive number of match entries.

Specifically, the `set_asymmetry_index` action computes the discretized asymmetry bin and assigns it to the metadata field `meta.curr_asymmetry_index_bin`. A similar assignment is performed for the previous asymmetry bin corresponding to AI_{t-2}. The reward table then matches on the tuple composed of the previous action a_{t-1}, the current asymmetry bin, the previous asymmetry bin, and the current service degradation counter $ServDegr_t$.

The associated `set_reward` action assigns the immediate reward value according to the logic defined in Eq. (5.21) and the service degradation constraint.

Finally, two auxiliary tables support the computation of learning terms required by Q-learning. The *discount_table* applies the discount factor to the maximum estimated next-state Q-value via the *apply_discount* action, while the *learning_rate_table* applies the learning rate to the temporal-difference error through the *apply_learning_rate* action.

Together, these tables enable the decomposition of the Q-learning update into match-action operations that are compatible with the constraints of the P4 data plane.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    action set_bin(bit<3> bin){
        meta.bin = bin;
    }

    table in_pkt_count_bin {
        key = {
            meta.in_pkt_count: range;
        }
        actions = {
            set_bin;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    table out_pkt_count_bin {
        key = {
            meta.out_pkt_count: range;
        }
        actions = {
            set_bin;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    table in_byte_count_bin {
        key = {
            meta.in_byte_count: range;
        }
        actions = {
            set_bin;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    table out_byte_count_bin {
        key = {
            meta.out_byte_count: range;
```

```
48          }
49          actions = {
50              set_bin;
51              NoAction;
52          }
53          size = 1024;
54          default_action = NoAction();
55      }
56
57      action set_amplification_factor_byte(bit <16> amp_byte){
58          meta.amplification_factor_byte = amp_byte;
59      }
60
61      action set_amplification_factor_pkt(bit <16> amp_pkt){
62          meta.amplification_factor_pkt = amp_pkt;
63      }
64
65      table amplification_factor_byte {
66          key = {
67              meta.f2: exact;
68              meta.f3: exact;
69          }
70          actions = {
71              set_amplification_factor_byte;
72              NoAction;
73          }
74          size = 1024;
75          default_action = NoAction();
76      }
77
78      table amplification_factor_pkt {
79          key = {
80              meta.f0: exact;
81              meta.f1: exact;
82          }
83          actions = {
84              set_amplification_factor_pkt;
85              NoAction;
86          }
87          size = 1024;
88          default_action = NoAction();
89      }
90
91      action set_asymmetry_index(bit<8> ai, bit<5> ai_bin){
92          meta.asymmetry_index = ai;
93          meta.curr_asymmetry_index_bin = ai_bin;
94      }
95
96      table asymmetry_index {
97          key = {
98              meta.amplification_factor_byte: exact;
```

```
99              meta.amplification_factor_pkt: exact;
100             meta.port_to_check: exact;
101         }
102         actions = {
103             set_asymmetry_index;
104             NoAction;
105         }
106         size = 1024;
107         default_action = NoAction();
108     }
109
110     action set_reward(int<8> reward, bit<1> negative){
111         meta.reward = reward;
112         is_negative = (negative == 1) ? 1w1 : 1w0;
113     }
114
115     table reward {
116         key = {
117             meta.curr_asymmetry_index_bin: exact; // AI_{t-1}
118             meta.curr_action: exact; // a_{t-1}
119             meta.prev_asymmetry_index_bin: exact; // AI_{t-2}
120             meta.service_degradation: exact; // ServDegr_t
121         }
122         actions = {
123             set_reward; // write meta.reward = R(s,a)
124             NoAction; // leave reward unchanged if no match
125         }
126         size = 32768;
127         default_action = NoAction();
128     }
129
130     action apply_discount(int<QVALUE_BITS_PER_ACTION> q_value, bit<1>
        negative) {
131         discounted_q = q_value;
132         //is_negative = (negative == 1) ? 1w1 : 1w0;
133         if(negative == (bit<1>)1){
134             is_negative = (bit<1>)1;
135         }
136         else{
137             is_negative = (bit<1>)0;
138         }
139     }
140
141     table discount_table {
142         key = {
143             max_next_q: exact; // highest Q-value from the next state
144         }
145         actions = {
146             apply_discount;
147             NoAction;
148         }
```

```
149         size = 1024;
150         default_action = NoAction();
151     }
152
153     action apply_learning_rate(int<QVALUE_BITS_PER_ACTION> q_value) {
154         learned_q = q_value;
155     }
156
157     table learning_rate_table {
158         key = {
159             td_error_reg: exact; // the temporal-difference error δ
160         }
161         actions = {
162             apply_learning_rate;
163             NoAction;
164         }
165         size = 1024;
166         default_action = NoAction();
167     }
```

Listing 6.5: Egress feature discretization, amplification metrics, and reward computation tables

The Egress apply block in Listing 6.6 implements the complete reinforcement learning workflow, including action selection, Q-value update, and policy enforcement. This block represents the operational core of the system, as it translates aggregated traffic features into adaptive mitigation decisions directly within the data plane.

At the beginning of the block, a pseudo-random value is generated and updated using the packet ingress timestamp. This value is later used to implement an ϵ-greedy exploration strategy, enabling probabilistic switching between exploitation of the learned policy and exploration of alternative actions, despite the absence of true randomness in P4 (lines 5-11). For flows associated with amplifiable services, the previously selected action is retrieved to ensure consistent enforcement across packets.

When the ingress pipeline signals that a new decision window has completed, the egress block performs two main operations: feature discretization and state reconstruction for Q-table indexing.

First, per-flow traffic statistics are discretized into compact binned features. Incoming and outgoing packet counts, as well as incoming and outgoing byte counts, are mapped into predefined bins through dedicated match-action tables. The resulting bin identifiers are stored in metadata fields (e.g., `meta.f0`–`meta.f3`).

Second, the egress pipeline reconstructs the history-aware reinforcement learning state used for reward computation and Q-table access. The byte-level and packet-level amplification factors are computed, followed by the asymmetry index, which result is stored in `meta.curr_asymmetry_index_bin`. The remaining elements of the state, namely AI_{t-2}, a_{t-1}, a_{t-2}, and $ServDegr_t$, are read from dedicated per-flow registers.

These components are then combined into a compact Q-table index through bitwise concatenation. Starting from the previous asymmetry bin, the index is progressively shifted and augmented with the previous action, the current asymmetry bin, the current action, and the service degradation value. This bit-level composition yields a unique integer index representing the state tuple

$$(AI_{t-2}, a_{t-2}, AI_{t-1}, a_{t-1}, ServDegr_t) \tag{6.1}$$

which is subsequently used to access and update the Q-table. The corresponding Q-table entry is then read and unpacked into per-action Q-values. Uninitialized entries are lazily initialized on first access to avoid pre-populating the table externally (lines 56-68). This design enables efficient in-switch reinforcement learning while preserving the temporal structure required by the history-aware state formulation.

Action selection follows an ϵ-greedy policy: if exploration is triggered or the action history is still incomplete, a random action is selected; otherwise, the action with the highest estimated Q-value is chosen. Once the action history is fully populated and the training phase is still active, the block proceeds with the learning step.

An immediate reward is then obtained by matching the discretized asymmetry history, the previously taken action, and the current service degradation state.

The Q-learning update is then performed entirely in the data plane. This includes computing the discounted estimate of future reward, deriving the temporal-difference error, applying the learning rate, and updating only the Q-value corresponding to the previously selected action. Careful handling of signed values, saturation bounds, and offset encoding ensures correctness within the constraints of P4 arithmetic. For brevity, the detailed implementation logic for signed arithmetic handling and boundary checks is omitted here; the complete implementation is available in the accompanying GitHub repository [25].

The updated Q-values are repacked and written back to the Q-table. The learning window counter is then incremented, and the current Q-index is stored for the next iteration. Subsequently, the action history is updated through bitwise shifting to retain the most recent decisions, and the current asymmetry bin is stored as the previous asymmetry for the next window. Finally, the service degradation counter is adjusted according to the selected action, and all updated state variables are written back to their corresponding registers (lines 138-189).

Finally, the selected action is enforced: if the flow corresponds to an amplifiable service and the chosen action is DROP, the packet is discarded; otherwise, it is allowed to exit the pipeline.

```
1  apply {
2          // Read the current pseudo-random byte from the register.
3          // This byte will be used to implement ϵ-greedy behavior.
4          rand_byte_reg.read(rand_byte, 0);
5
6          // Update the pseudo-random byte by XOR-ing it with the lower
               8 bits of the current packet's ingress timestamp.
7          rand_byte = rand_byte ^
               standard_metadata.ingress_global_timestamp[7:0];
```

```
        rand_byte_reg.write(0, rand_byte);

        if(meta.is_amplifiable_service == 1){
            action_to_take_reg.read(action_to_take, meta.flow_id);
        }

        // Only trigger RL decision when flagged by MyIngress
        if(meta.agent_trigger == ACTIVE){

            // Read the number of completed time windows for this flow
            time_windows_reg.read(time_windows, meta.flow_id);

            // 1) Feature binning: discretize counts
            in_pkt_count_bin.apply(); // Bin incoming packet count
            meta.f0 = meta.bin; // Save as f0

            out_pkt_count_bin.apply(); // Bin outgoing packet count
            meta.f1 = meta.bin; // Save as f1

            in_byte_count_bin.apply(); // Bin incoming byte count
            meta.f2 = meta.bin; // Save as f2

            out_byte_count_bin.apply(); // Bin outgoing byte count
            meta.f3 = meta.bin; // Save as f3

            // 2) Compute Q-table index from
            (AI_{t-1}, a_{t-1}, AI_{t-2}, a_{t-2}, ServDegr_t)

            // Compute byte-level amplification factor
            amplification_factor_byte.apply();

            // Compute packet-level amplification factor
            amplification_factor_pkt.apply();

            // Compute asymmetry index (formula set via control plane)
            asymmetry_index.apply();

            // Retrieve the last actions taken for this flow
            last_actions_taken_reg.read(last_actions_taken,
                meta.flow_id);

            // a_{t-1} = MSB
            meta.curr_action = (bit<1>)(last_actions_taken >>
                (LAST_ACTIONS_TAKEN_BITS - 1));

            // a_{t-2} = LSB
            meta.prev_action = (bit<1>) last_actions_taken;

            // ServDegr_t
            service_degradation_reg.read(meta.service_degradation,
                meta.flow_id);
```

```
56
57          // AI_{t-2}
58          prev_asymmetry_index_bin_reg.read(meta.prev_asymmetry_index_bin,
                meta.flow_id);
59
60          bit<32> idx;
61
62          idx = (bit<32>)meta.prev_asymmetry_index_bin;
63          idx = (idx << 1) | (bit<32>)meta.prev_action;
64          idx = (idx << 5) | (bit<32>)meta.curr_asymmetry_index_bin;
65          idx = (idx << 1) | (bit<32>)meta.curr_action;
66          idx = (idx << 4) | (bit<32>)meta.service_degradation;
67
68          // 3) Read packed Q-values (two 8-bit values) at index `idx`
69          // The lower Q_VALUES_BITS_PER_ACTION bits represent
                Q-value for action 1 (Q1),
70          // and the upper Q_VALUES_BITS_PER_ACTION bits represent
                Q-value for action 2 (Q2).
71          qtable.read(q, idx);
72
73          // Unpack Q1 (ALLOW) and Q2 (DROP), using bit slices
74          // Q-values are stored using offset encoding to support
                negative values.
75          q1 = q[LEFT_IDX_Q1:0]; // ALLOW
76          q2 = q[LEFT_IDX_Q2:RIGHT_IDX_Q2]; // DROP
77
78          // Lazy initialization:
79          // If both Q1 and Q2 are zero, assume the entry has never
                been initialized.
80          // We initialize both Q-values to Q_OFFSET, which encodes Q
                = 0 in offset logic.
81          if (q1 == 0 && q2 == 0) {
82              bit<QVALUE_BITS> init_val = ((bit<QVALUE_BITS>)Q_OFFSET
                    << SHIFT_OFFSET) | (bit<QVALUE_BITS>)Q_OFFSET;
83              qtable.write(idx, init_val);
84              q1 = (bit<QVALUE_BITS_PER_ACTION>)Q_OFFSET;
85              q2 = (bit<QVALUE_BITS_PER_ACTION>)Q_OFFSET;
86          }
87
88          // 4) ε-greedy action selection
89          // time_windows < LAST_ACTIONS_TAKEN_BIT : My state is
                still partial: I don't have a full action history yet.
90          // I perform pure exploration and store the chosen action
                and qvalue index.
91           if (time_windows < TRAINING_WINDOWS && (rand_byte <
                EPSILON || time_windows < LAST_ACTIONS_TAKEN_BITS) ) {
92              // Exploration: choose action by LSB of rand_byte
93              action_to_take = (bit<1>) rand_byte;
94          } else {
95              // Exploitation: pick max-valued action
```

```
96              action_to_take = (q2 > q1) ? (bit<1>)DROP :
                    (bit<1>)ALLOW;
97          }
98
99          // Track maximum next-state Q for TD-error and reward
100         max_next_q = (q2 > q1) ? q2 : q1;
101
102         // At this point, the state is complete: the action history
                is full. I can now safely access the Q-table and
                perform Q-value updates, if the training is not over
                yet.
103         if(time_windows < TRAINING_WINDOWS && time_windows >=
                LAST_ACTIONS_TAKEN_BITS){
104
105             // 5) Lookup reward based on the last action and
                    asymmetry index
106             reward.apply();
107
108             // 6) Read index of previous Q-value and the packed
                    Q-entry
109             prev_qvalue_idx_reg.read(prev_qvalue_idx, meta.flow_id);
110             bit<QVALUE_BITS> prev_q;
111             qtable.read(prev_q, prev_qvalue_idx);
112
113             // 7) Extract the Q-value for the action actually taken
                    last time
114             bit<QVALUE_BITS_PER_ACTION> prev_q_for_last_action;
115             if (meta.last_action_taken == ALLOW) {
116                 // If the last action was "ALLOW", grab the
                        low-order bits
117                 prev_q_for_last_action = prev_q[LEFT_IDX_Q1:0];
118             } else {
119                 // If it was "DROP", grab the high-order bits
120                 prev_q_for_last_action =
                        prev_q[LEFT_IDX_Q2:RIGHT_IDX_Q2];
121             }
122
123             // 8) Lookup discounted next-state value: γ · max
                    Q(s',a')
124             discount_table.apply();
125
126             int<QVALUE_BITS_PER_ACTION> real_prev_q_for_last_action
                    = prev_q_for_last_action - Q_OFFSET; // shift from
                    [0,255] to [-128, 127]
127
128             // 9) Compute TD-error δ = r_t + γ · max Q(s',a') -
                    Q(s,a)
129             int<QVALUE_BITS> td_error_reg = meta.reward +
                    discounted_q - real_prev_q_for_last_action;
130
131             // 10) Lookup learning rate α · TD-error
```

```
132                 learning_rate_table.apply();
133
134                 // 11) Split packed previous Q into per-action values
135                 bit<QVALUE_BITS_PER_ACTION> prev_q1 =
                        prev_q[LEFT_IDX_Q1:0];
136                 bit<QVALUE_BITS_PER_ACTION> prev_q2 =
                        prev_q[LEFT_IDX_Q2:RIGHT_IDX_Q2];
137
138                 // 12) Update only the Q-value corresponding to the
                        last action taken
139                 if (meta.last_action_taken == ALLOW) {
140                     prev_q1 = prev_q_for_last_action + learned_q;
141                 else{
142                     prev_q2 = prev_q_for_last_action + learned_q;
143                 }
144
145                 // 13) Pack updated Q1/Q2 back and write to Q-table
146                 prev_q = ((bit<QVALUE_BITS>)prev_q2 << SHIFT_OFFSET) |
                        (bit<QVALUE_BITS>)prev_q1;
147                 qtable.write(prev_qvalue_idx, prev_q);
148
149             }
150             // 14) Increase counter used to check if learning time is
                    over
151             time_windows = time_windows + 1;
152             time_windows_reg.write(meta.flow_id, time_windows);
153
154             // 15) Record last action and Q-index for next iteration
155             //last_action_taken_reg.write(meta.flow_id, action_to_take);
156             prev_qvalue_idx_reg.write(meta.flow_id, idx);
157
158             // Read old history
159             bit<LAST_ACTIONS_TAKEN_BITS> old = last_actions_taken;
160
161             // Shift right by one position to drop the oldest bit
162             bit<LAST_ACTIONS_TAKEN_BITS> shifted = old >> 1;
163
164             // Position the new action at the MSB
165             bit<LAST_ACTIONS_TAKEN_BITS> insert =
                    (bit<LAST_ACTIONS_TAKEN_BITS>)(action_to_take) <<
                    (LAST_ACTIONS_TAKEN_BITS - 1);
166
167             // Combine shifted history and insert via bitwise OR
168             last_actions_taken = shifted | insert;
169
170             // Save the updated action history to the register
171             last_actions_taken_reg.write(meta.flow_id,
                    last_actions_taken);
172
173             // Save the current asymmetry index to the register
```

```
174             prev_asymmetry_index_bin_reg.write(meta.flow_id,
                    meta.curr_asymmetry_index_bin);
175
176             // Update Service Degradation accordingly to the action
                    chosen
177             if(action_to_take == DROP){
178                 if(meta.service_degradation < SERVDEGR_MAX){
179                     meta.service_degradation += 1;
180                 }
181             }
182             else{
183                 if(meta.service_degradation > 0){
184                     meta.service_degradation -= 1;
185                 }
186             }
187             service_degradation_reg.write(meta.flow_id,
                    meta.service_degradation);
188
189             action_to_take_reg.write(meta.flow_id, action_to_take);
190         }
191
192         // 16) If this is an amplifiable service and action=DROP, then
                drop packet
193         if(meta.is_amplifiable_service & action_to_take == DROP ){
194             drop();
195         }
196         // else (action_to_take == ALLOW ) we let the packet exit the
                pipeline...
197     }
198 }
```

Listing 6.6: Egress apply logic for Q-learning update, ϵ-greedy action selection, and enforcement

6.1.4 Deparser and Pipeline instantiation

At the end of the implementation, the deparser and the top-level pipeline instantiation define how packets are reconstructed and forwarded to the egress interface. The checksum computation block is responsible for recomputing the IPv4 header checksum after packet processing, ensuring protocol correctness whenever header fields such as the time-to-live are modified during ingress or egress operations. This step guarantees that packets emitted by the switch remain compliant with standard IPv4 semantics.

The deparser control block specifies the order in which the extracted headers are reassembled and emitted onto the wire. In this implementation, packets are emitted with the same protocol structure as the incoming traffic, as no custom encapsulation or header insertion is performed.

Finally, the top-level switch instantiation defines the complete packet-processing

pipeline and the order in which packets traverse its components, reflecting the standard v1model architecture adopted in this work.

```
control MyComputeChecksum(inout headers hdr, inout metadata meta) {
     apply {
        update_checksum(
        hdr.ipv4.isValid(),
            { hdr.ipv4.version,
              hdr.ipv4.ihl,
              hdr.ipv4.diffserv,
              hdr.ipv4.totalLen,
              hdr.ipv4.identification,
              hdr.ipv4.flags,
              hdr.ipv4.fragOffset,
              hdr.ipv4.ttl,
              hdr.ipv4.protocol,
              hdr.ipv4.srcAddr,
              hdr.ipv4.dstAddr },
            hdr.ipv4.hdrChecksum,
            HashAlgorithm.csum16);
    }
}

control MyDeparser(packet_out packet, in headers hdr) {
    apply {
        packet.emit(hdr.ethernet);
        packet.emit(hdr.ipv4);
        packet.emit(hdr.udp);
    }
}

// Top-level switch instantiation
V1Switch(
MyParser(),
MyVerifyChecksum(),
MyIngress(),
MyEgress(),
MyComputeChecksum(),
MyDeparser()
) main;
```

Listing 6.7: Deparser, checksum recomputation, and top-level pipeline definition

6.2 Control Plane

The match-action tables in the programmable data plane are initialized through an external **Python** script that acts as a lightweight control-plane component using the P4Runtime API. The script establishes a gRPC connection with the switch and programmatically configures the match-action tables exposed by the data plane.

Rather than statically defining table entries, the script generates them algorithmically based on predefined configuration parameters and reinforcement learning constants, enabling reproducible initialization of the switch state.

Table entries are constructed in software and translated into P4Runtime *TableEntry* messages, which are then written to the switch at startup. This design choice also provides significant flexibility, as key components of the learning model can be modified or tuned directly in the control plane without requiring changes to the data plane program. In particular, the control plane script (`controller.py`), available in the project GitHub repository [25], allows easy reconfiguration of several key parameters, including:

- **Discretization parameters**, such as the range boundaries used to map packet and byte counters into discrete state features, as well as the minimum and maximum bin identifiers for the asymmetry index.
- **Amplification factor computation**, including the functional mapping between incoming and outgoing traffic bins and the resulting amplification values for both packet- and byte-level metrics.
- **Asymmetry index formulation**, defined through tunable weighting coefficients that balance the contribution of packet-based and byte-based amplification factors.
- **Reward function configuration**, including the bin-level thresholds corresponding to the symbolic parameters β_L, β_M^-, β_M^+, and β_H, which determine the boundaries between low, moderate, and high asymmetry regions. These thresholds are specified in terms of asymmetry bins, allowing flexible adjustment without modifying the data plane logic.
- **Service degradation parameters**, such as the degradation threshold and maximum counter value.
- **Reinforcement learning hyperparameters**, such as the learning rate and discount factor, which influence the convergence behavior of the in-switch learning process.
- **Service-specific parameters**, allowing different amplification-sensitive services to be weighted or treated differently without modifying the data plane logic.

```
def load_and_write_table_entries(entries, p4info_helper, sw):
    # Install table entries on the switch via P4Runtime.
    for entry in entries:
        table_name = entry["table"]
        action_name = entry["action_name"]
        action_params= entry.get("action_params", {})
        default_act = entry.get("default_action", False)
        priority = entry.get("priority", None)
        match_fields = {}

```

```
        # Convert JSON match spec to P4Runtime format
        for field, val in entry.get("match", {}).items():
            if isinstance(val, list) and isinstance(val[0], str):
                match_fields[field] = (val[0], val[1])
            elif isinstance(val, list) and all(isinstance(x, int) for x
                in val):
                match_fields[field] = (val[0], val[1])
            else:
                match_fields[field] = val

        # Build and write the TableEntry
        te = p4info_helper.buildTableEntry(
            table_name = table_name,
            match_fields = match_fields if match_fields else None,
            action_name = action_name,
            action_params = action_params,
            default_action = default_act,
            priority = priority
        )
        sw.WriteTableEntry(te)

def main(p4info_file_path, bmv2_file_path):
    # Instantiate a P4Runtime helper from the p4info file
    p4info_helper = p4runtime_lib.helper.P4InfoHelper(p4info_file_path)

    try:
        # Create a switch connection object for s1;
        # this is backed by a P4Runtime gRPC connection.
        # Also, dump all P4Runtime messages sent to switch to given
            txt files.
        s1 = p4runtime_lib.bmv2.Bmv2SwitchConnection(
            name='s1',
            address='127.0.0.1:50051',
            device_id=0,
            proto_dump_file='logs/s1-p4runtime-requests.txt')

        # Send master arbitration update message to establish this
            controller as master (required by P4Runtime before
            performing any other write operation)
        s1.MasterArbitrationUpdate()

        # Install the P4 program on the switches
        s1.SetForwardingPipelineConfig(p4info=p4info_helper.p4info,
                                       bmv2_json_file_path=bmv2_file_path)

        # Build and load all entries
        entries = build_all_entries()
        load_and_write_table_entries(entries, p4info_helper, s1)
```

Listing 6.8: P4Runtime-based table population and pipeline installation

Chapter 7

Experimental Evaluation

This chapter provides an in-depth technical description of the experimental evaluation conducted to assess the effectiveness of the proposed mitigation approach. The primary objective of the experiments is to verify whether the system can effectively adapt in real time to evolving traffic patterns and identify malicious flows directly within the programmable data plane. In addition to evaluating the behavior of the reinforcement learning-based mitigation strategy, the results are compared against a commonly adopted baseline approach based on port-based filtering, which represents a simple and widely used method in firewalls for mitigating amplification-based attacks [26].

The remainder of this chapter is structured as follows. First, the experimental setup is described, including the hardware and software environment used to run the experiments. Next, the experimental workflow is presented, detailing how traffic scenarios are generated and how the system interacts with the network during training and evaluation. Finally, the results obtained under different experimental scenarios are presented and discussed, highlighting the behavior of the proposed approach and comparing it with the baseline mitigation strategy.

7.1 Experimental Setup

All experiments were conducted using the official open-source P4 development environment provided by the P4 Tutorials project [27]. The source code of the prototype is publicly available [25]. The environment consists of a preconfigured virtual machine based on **Ubuntu 24.04 Desktop (AMD64)** that includes the main P4 software components compiled from source as of **March 1st, 2025**. The VM integrates the P4 compiler (`p4c`), the Behavioral Model v2 (BMv2) software switch, the Mininet network emulator, and the P4Runtime control interface, providing a complete platform for developing and evaluating programmable data plane applications. The virtual machine was executed with the configuration in Table 7.1.

The experimental environment is compiled and executed through the standard build process provided by the P4 Tutorials framework. By running the `make` command in the project directory, the P4 source code is first compiled using the `p4c`

Resource	Configuration
CPU	2 virtual cores
RAM	4096 MB
Disk	60 GB
Operating System	Ubuntu 24.04 Desktop (AMD64)

Table 7.1: Hardware configuration of the experimental environment.

compiler, which generates the `simple_switch_grpc` target configuration files required by the BMv2 switch, which exposes both a Thrift interface and a gRPC-based P4Runtime API.

Once the compilation phase is completed, the build process automatically instantiates the network topology defined in the `topology.json`, which describes the structure of the virtual network, including the set of hosts, the programmable switch instance, and the links interconnecting them. This description is parsed by the topology construction scripts, which automatically instantiate the corresponding network within the **Mininet** emulator.

Mininet dynamically creates a virtual network composed of software hosts, links, and the BMv2 switch instance. Each host is implemented as a Linux network namespace with its own network stack, effectively behaving as an isolated network node connected to the programmable switch through virtual Ethernet interfaces.

The topology used in our experiments includes hosts representing the *victim*, the *attacker*, and two legitimate servers acting as reflectors in the amplification scenario (DNS and NTP), as depicted in Figure 5.1.

After the switch instance has been launched by Mininet, the control plane program (`controller.py`) connects to the switch through the P4Runtime gRPC interface and dynamically installs the necessary match-action rules and data-plane configurations required by the system. This initialization step ensures that the forwarding logic and reinforcement learning components are correctly configured before the experiments begin.

Once the environment is fully initialized, the Mininet command-line interface becomes available, allowing direct interaction with the emulated network. From this interface it is possible to execute commands on individual hosts to monitor the behavior of the programmable switch during the experimental evaluation and to generate traffic flows using Scapy-based Python scripts developed for the experiments. In particular, the scripts `victim.py`, `attacker.py`, and `server.py` emulate the behavior of the victim, the attacker, and the reflector servers, respectively, and are available in the public repository accompanying this work [25]. The traffic generation model implemented in these scripts is described in Section 7.2.

7.2 Experimental Workflow and Assumptions

The traffic generation model used in the experiments aims to emulate realistic amplification-based attack conditions while maintaining controlled and reproducible experimental scenarios. To this end, the attack behavior alternates between two main traffic patterns: **cyclic attacks** and **burst attacks**.

The cyclic attack pattern represents a gradual escalation scenario in which the attack intensity increases progressively over consecutive time windows (e.g., from low to medium to high), remains at a high plateau for a certain period, and then decreases symmetrically. Such temporally structured behaviors have been discussed in the literature on pulsing and low-rate denial-of-service attacks, where attackers intentionally modulate traffic intensity over time to probe defenses and sustain service degradation [28].

In contrast, the burst attack pattern captures sudden and short-lived traffic spikes characterized by an abrupt transition to medium or high intensity for a limited number of windows. Similar traffic surges have been observed in large-scale DDoS campaigns launched by botnets such as Mirai, which generated massive volumetric floods against high-profile targets [29].

By alternating these two patterns and interleaving them with short silent phases, the resulting traffic profile generates heterogeneous and temporally structured conditions. This design allows the learning agent to experience both gradual pressure buildup and abrupt overload situations during training.

To expose the agent to a wide variety of traffic conditions, the attack sequence is generated through a randomized procedure. However, to ensure full experimental reproducibility, the pseudo-random generation process is controlled using fixed seeds. By specifying the seed value for each run, the same sequence of attack windows can be deterministically reconstructed, enabling consistent comparisons across different parameter configurations and training sessions.

Regarding the amplification behavior of the reflector servers, the **BAF** and **PAF** values are inspired by the empirical measurements reported by Rossow [12] in his seminal study on amplification-based DDoS attacks, which are reported in Table 2.1. This work provides experimentally observed amplification ranges for several protocols, including DNS and NTP, which are exploited in the experimental scenarios considered in this thesis, thus ensuring that the generated traffic reflects realistic amplification dynamics.

For the **legitimate traffic** generated by the victim, protocol-specific packet sizes are also modeled according to realistic observations. In the case of NTP, both request and response packets have a fixed size of 48 bytes, consistent with the standard NTP message format. For DNS traffic, instead, the size of the response depends on the corresponding DNS query. To model this variability, we rely on measurements reported by Foremski et al. [30], which provide statistical insights into DNS traffic characteristics derived from large-scale passive measurements of real-world DNS deployments. These statistics capture benign production DNS behavior and report realistic distributions of DNS response sizes and frequencies based on observed packet traces. Using these measurements allows the traffic generator

to emulate DNS traffic patterns that closely resemble those observed in operational networks.

The exact parameters and packet size values used in the traffic generator are implemented in the `server.py` module of the public repository accompanying this work [25].

Each experiment is organized into **discrete observation windows of 10 seconds**. The traffic generation process varies across time windows and allows the attacker to operate at four different intensity levels, namely *SILENT*, *LOW*, *MEDIUM*, and *HIGH*. The victim, instead, generates legitimate traffic only at *LOW*, *MEDIUM*, and *HIGH* levels. The *SILENT* state is not considered for the victim because it could result in observation windows where both the victim and the attacker generate no traffic, providing no useful information for the learning process. Traffic intensity levels correspond to different packet generation rates within each observation window, thereby emulating varying network load conditions during the experiment.

Traffic level	**Victim packets/window**	**Attacker packets/window**
SILENT	X	0
LOW	10	10
MEDIUM	30	30
HIGH	50	75

Table 7.2: Traffic intensity levels used in the experiments.

The victim traffic follows a probabilistic state-based model. Before the experiment starts, a sequence of traffic states is generated according to a predefined distribution: *LOW* (0.3), *MEDIUM* (0.4), and *HIGH* (0.3). Each time window then follows the corresponding state in this pre-generated sequence.

The attack traffic follows a state-driven profile designed to emulate realistic amplification attack dynamics. As discussed earlier, the attack model alternates between cyclic escalation phases and short burst events, which are interleaved with silent periods. The detailed configuration used in the experiments is summarized as follows:

- **Silent phase:** no attack traffic is generated for a duration randomly sampled between 3 and 5 windows.
- **Cyclic attack:** as described above, the attack intensity gradually increases from LOW to MEDIUM, reaches a HIGH plateau lasting between 3 and 4 windows, and then decreases symmetrically back to LOW.
- **Burst attack:** as previously discussed, a sudden spike of attack traffic occurs at either MEDIUM or HIGH intensity and lasts between 4 and 6 windows.

The **learning rate** (α) and **discount factor** (γ) were selected according to commonly adopted values in reinforcement learning literature. In particular, the discount factor was set to $\gamma = 0.9$, a standard choice that balances short-term

and long-term rewards by prioritizing future returns while still maintaining responsiveness to immediate feedback. This value is appropriate for our scenario, where defensive actions may have delayed consequences on service degradation and attack mitigation effectiveness. The learning rate was set to $\alpha = 0.1$, a value that allows the agent to progressively update its knowledge while avoiding abrupt oscillations in the estimated Q-values.

The **training phase** was conducted over 2500 time windows. Due to constraints in traffic generation, the adopted traffic model produces roughly 3000 reachable states, many of which occur rarely and do not require extensive exploration. Since each window lasts 10 seconds, increasing the number of training windows would significantly extend the duration of the experiments. Therefore, 2500 windows were selected as a practical compromise between learning effectiveness and experimental feasibility while still demonstrating the viability of reinforcement learning for in-switch DDoS mitigation. After the training phase, an additional set of 500 windows is used as a **test phase**, during which the agent's behavior is evaluated and the experimental results are collected, resulting in a total duration of approximately 30000 seconds (about 8.3 hours) per scenario.

Finally, the **service degradation threshold** X is set to 8. This choice is motivated by the characteristics of the adopted traffic model, where the maximum possible duration of an attack phase N is eight consecutive windows. Consequently, when the service degradation exceeds this threshold, the agent is penalized, encouraging policies that prevent sustained service degradation over prolonged attack periods.

For each evaluated protocol (DNS and NTP), a series of experimental scenarios is considered by varying two key parameters: the **weighting coefficients** w_{b} and w_{p} used in the computation of the Asymmetry Index and the **exploration rate** (ϵ) of the reinforcement learning agent. The same traffic generation model previously explained is used across all evaluated scenarios. Each scenario corresponds to a specific combination of protocol and parameter configuration, as summarized in Table 7.3.

The weight parameters directly influence the relative contribution of packet-level and byte-level metrics within the AI formulation 5.15, thereby affecting the agent's perception of traffic imbalance and attack intensity. By systematically adjusting these coefficients, we assess the sensitivity of the learning process to different feature emphases and evaluate how the agent adapts to different interpretations of traffic asymmetry.

In addition, different exploration rates are tested to analyze the impact of exploration on policy learning. In particular, a higher exploration rate ($\epsilon = 0.3$) was included to encourage sufficient exploration during the early training stages, when Q-values are still close to zero. In our implementation, when action-value estimates are equal, the default selection mechanism favors the `ALLOW` action. Consequently, increasing ϵ raises the probability of selecting the `DROP` action even under nominal traffic conditions. This allows the agent to experience and learn from overreaction states (i.e., unnecessary packet drops), ultimately improving its ability to distinguish between benign traffic asymmetries and genuine attack scenarios.

Scenario	Protocol	w_b	w_p	ϵ
1.1	DNS	0.5	0.5	0.1
1.2	DNS	0.6	0.4	0.1
1.3	DNS	0.6	0.4	0.3
2.1	NTP	0.5	0.5	0.1
2.2	NTP	0.4	0.6	0.1
2.3	NTP	0.5	0.5	0.3

Table 7.3: Experimental scenarios evaluated in the experimental campaign.

7.3 Results and Discussion

7.3.1 DNS Amplification Attack Scenario

Figure 7.1 shows the evolution of the average reward during training for the DNS scenarios, while Figure 7.2 reports the detection rate achieved for different attacker profiles (*SILENT*, *LOW*, *MEDIUM*, and *HIGH*). The **detection rate** represents the percentage of time windows in which the mitigation decision taken by the agent correctly matches the actual attacker activity level observed in that window. In particular, windows corresponding to active attack profiles (*LOW*, *MEDIUM*, and *HIGH*) are considered correctly detected when the agent selects the `DROP` action, whereas *SILENT* windows are considered correctly classified when the agent selects `ALLOW`. In practical terms, this metric reflects the ability of the mitigation policy to block malicious traffic while preserving legitimate traffic.

The reward values shown in the training plots represent the cumulative average reward computed over fixed intervals of 100 time windows. This aggregation smooths short-term fluctuations caused by individual decisions and provides a clearer view of the overall learning trend of the agent during training.

Overall, the training curves show a stable learning behavior across all tested configurations. In all scenarios, the average reward gradually increases from slightly negative values and stabilizes around small positive values as the training progresses. This trend indicates that the agent progressively learns a policy that yields better long-term outcomes.

Regarding the reward evolution in Figure 7.1 during training, scenarios 1.1 and 1.2 exhibit a more stable learning behavior compared to scenario 1.3. This difference is mainly attributable to the exploration rate. In scenario 1.3, the higher exploration rate implies that, even in the later stages of training, the agent follows the learned policy with a lower probability compared to the configuration with $\epsilon = 0.1$. As a result, while $\epsilon = 0.3$ encourages broader exploration and can improve robustness against misleading early experiences, it also introduces additional randomness in the agent's behavior once the policy has largely converged, thus producing slightly larger oscillations in the training curve. Nevertheless, the overall reward trend remains positive, suggesting that exploration does not significantly degrade learning performance.

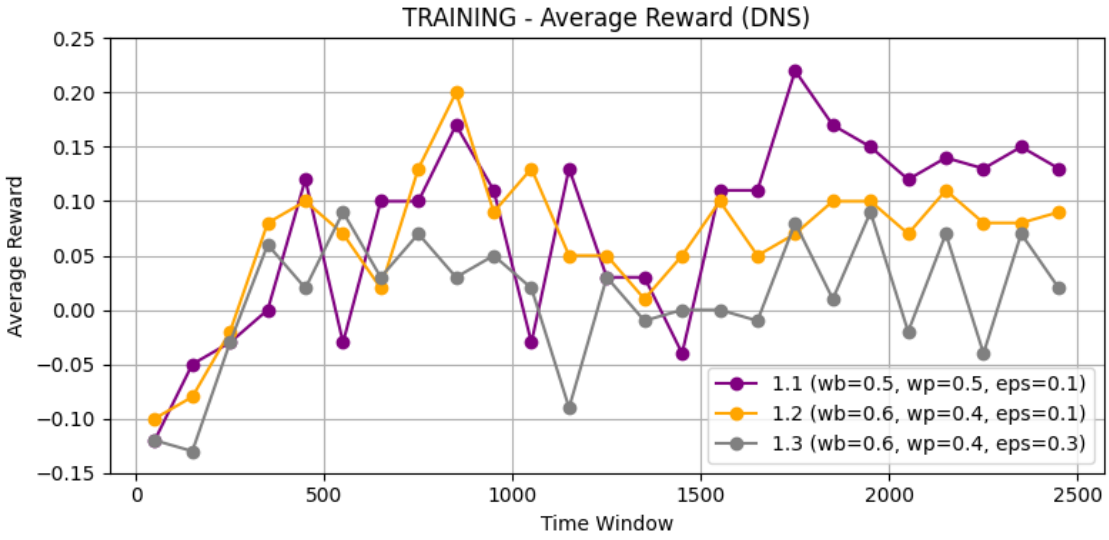

Figure 7.1: Training reward under different parameter configurations for the DNS attack scenario, shown as the average reward computed over intervals of 100 windows.

Comparing scenarios 1.1 and 1.2 in Figure 7.1, both configurations converge to similar reward levels, although scenario 1.1 stabilizes at a slightly higher value. This behavior is explained by the different frequency with which the service degradation threshold is reached during training. In scenario 1.1 the threshold is exceeded only rarely, meaning that the corresponding penalty term is activated infrequently. In contrast, during scenario 1.2 the service degradation metric surpasses the threshold more often, triggering additional negative rewards. As a result, the average reward observed during training is slightly lower, reflecting the increased number of penalized states encountered by the agent.

The detection statistics reported in Figure 7.2 further confirm the effectiveness of the learned policy. In the SILENT case, the agent correctly allows legitimate traffic in approximately 89–90% of the windows across the tested scenarios, indicating that the system rarely blocks benign traffic. For attack conditions, the detection performance increases with the attack intensity. In particular, the agent achieves detection rates above 74–80% for HIGH attack profiles and above 68–84% for MEDIUM attacks, demonstrating a strong ability to identify and mitigate significant amplification traffic.

As expected, detection performance is lower for LOW-intensity attacks, with rates ranging between approximately 23% and 36%. These attacks generate traffic patterns that are closer to legitimate request-response behavior, making them more difficult to distinguish using the asymmetry-based features available in the data plane. However, this conservative behavior also helps reduce false positives, which would otherwise negatively impact legitimate traffic.

This outcome can also be considered desirable, as LOW-intensity activity may correspond to probing phases in which the attacker tests the target or the mitigation system before launching a larger attack. Aggressively blocking traffic during these early stages could unnecessarily disrupt legitimate requests. By reacting more

cautiously in these situations, the agent avoids excessive mitigation and preserves service availability.

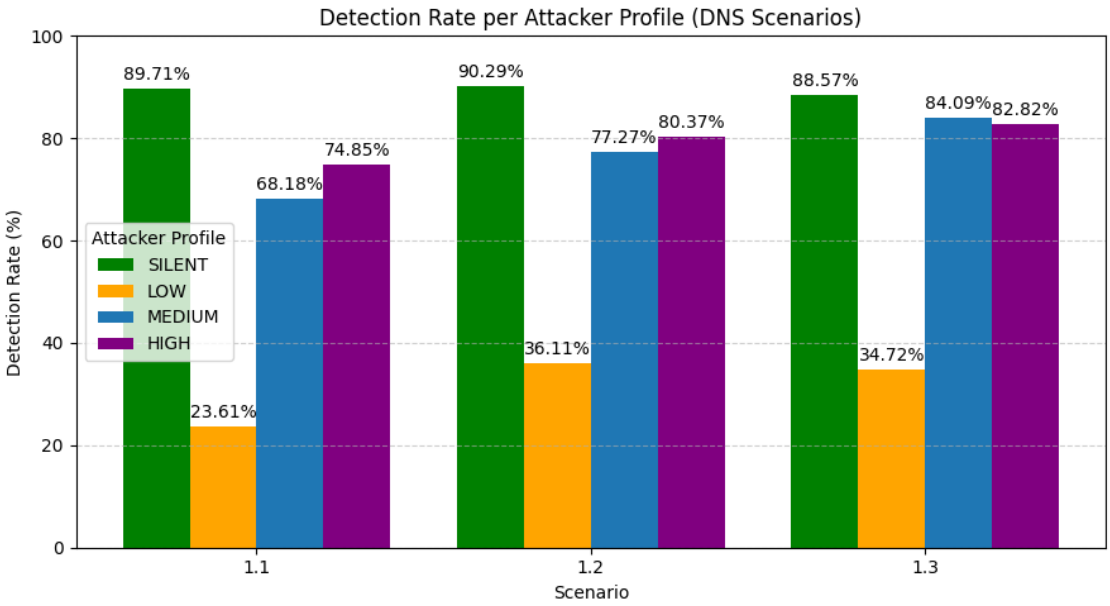

Figure 7.2: Detection rate of the proposed system across the DNS attack scenarios (1.1–1.3), reported for each attacker activity profile (SILENT, LOW, MEDIUM, HIGH). The results show how the detection performance varies with the intensity of the attack traffic.

Comparing the different parameter configurations, the detection statistics shown in the bar plots in Figure 7.2 further highlight the impact of the parameter configurations. In particular, scenarios 1.2 and 1.3 achieve the best detection performance. This observation is consistent with the characteristics of DNS amplification attacks. Since DNS responses are typically much larger than the corresponding requests, the asymmetry between inbound and outbound traffic is more pronounced in terms of transferred bytes rather than packet counts. Consequently, configurations that assign a higher relative importance to byte-level asymmetry provide better discrimination between legitimate and malicious traffic.

Returning to the service degradation aspect, the results obtained in scenario 1.3 are slightly higher than those observed in scenario 1.2. This difference can be attributed to the learning behavior induced by the service degradation penalty. In scenario 1.2, the agent learns to account for this penalty and progressively adapts its policy to avoid excessive traffic blocking once the service degradation threshold is reached. As a result, the agent tends to favor the `ALLOW` action in borderline situations, which slightly reduces the detection rate but contributes to preserving service availability.

To illustrate the impact of the service degradation component on the agent's decisions, Figure 7.3 shows a representative snapshot of the test phase for scenario 1.2. The figure reports three aligned plots describing the system state over a sequence of test windows.

The first plot shows the value of the AI observed in each window, together with the action selected by the agent. Green markers indicate windows in which the agent chooses the `ALLOW` action, while red markers represent windows in which the agent performs `DROP`. The second plot reports the traffic composition of each window. In this plot, the number of stacked rectangles represents the traffic intensity generated during that window, where 0 corresponds to SILENT, 1 to LOW, 2 to MEDIUM, and 3 to HIGH. The color identifies the traffic source: orange rectangles correspond to the victim's legitimate traffic, while purple rectangles represent the attack traffic. Finally, the third plot shows the evolution of the service degradation metric over time.

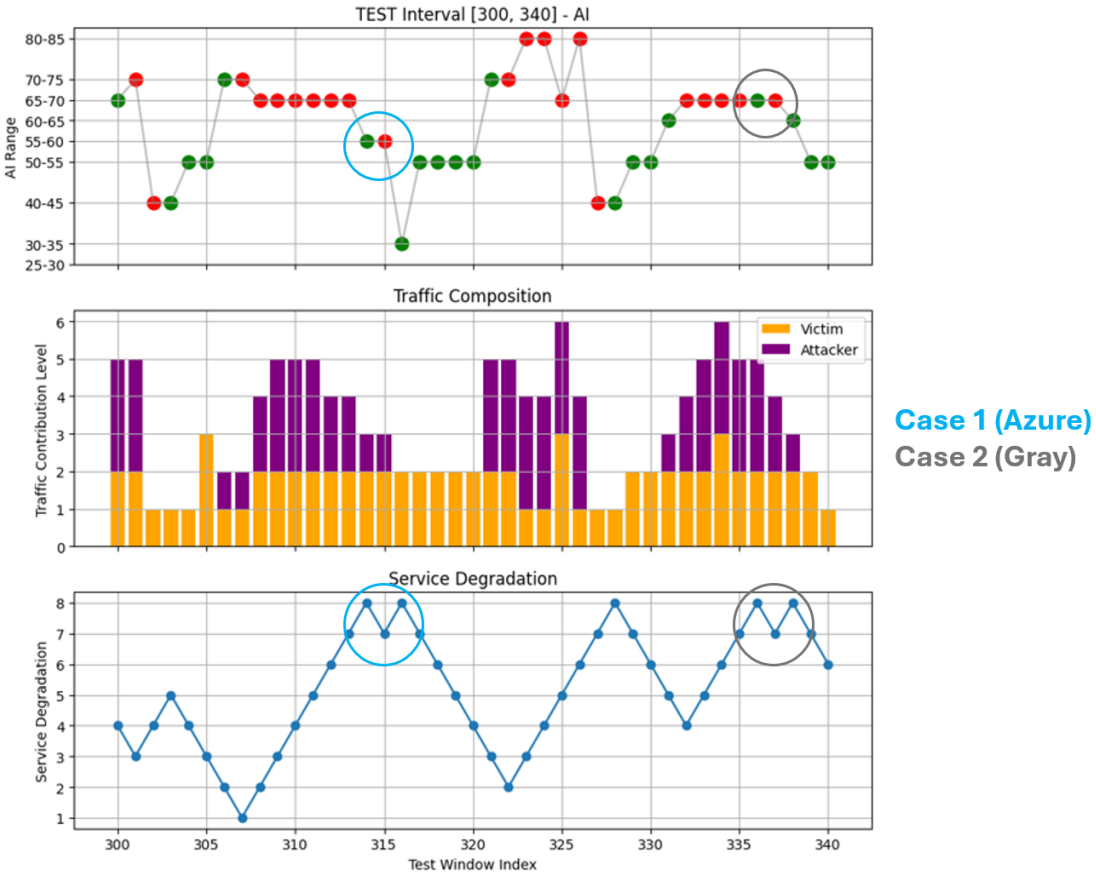

Figure 7.3: Example of agent decisions during the test interval for scenario 1.2, the top plot reports the Asymmetry Index observed by the agent and the corresponding action taken. The middle plot shows the traffic composition in each window, indicating the contribution levels of the victim and the attacker. The bottom plot reports the resulting service degradation experienced by the victim.

Case 1 (azure circle). Around window 314, the service degradation metric reaches the predefined threshold during the descending phase of a cyclic attack. In this window the attacker generates traffic with *LOW* intensity. Differently from the previous windows, in which the agent consistently applied the `DROP` action, the agent chooses `ALLOW` (green marker) in order to avoid exceeding the service degradation threshold. This behavior can be observed in the azure circle in the AI plot. In the following window the attack is still ongoing, and the agent resumes

dropping packets. As a result, the service degradation temporarily reaches a value of 8 again, but subsequently decreases as the attack phase ends.

Case 2 (gray circle). A similar behavior can be observed during another cyclic attack phase. In this case, when the service degradation metric approaches the threshold, the agent again opts for the `ALLOW` action despite the presence of a *MEDIUM*-intensity attack. This decision prevents the degradation metric from exceeding the predefined limit, illustrating how the reward formulation encourages the agent to balance attack mitigation with service availability.

7.3.2 NTP Amplification Attack Scenario

Figure 7.4 reports the evolution of the average reward during training for the NTP scenarios, while Figure 7.6 summarizes the detection performance for the different attacker profiles.

The reward curves show a stable learning trend across all tested configurations. In each scenario the average reward initially starts from slightly negative values and gradually increases as the training progresses, eventually stabilizing around small positive values, following the same trend observed in the DNS experiments.

Comparing the different configurations, the scenarios with $\epsilon = 0.1$ (2.1 and 2.2) exhibit a smoother and more stable convergence behavior (Figure 7.4). In contrast, scenario 2.3 ($\epsilon = 0.3$) presents larger oscillations throughout the training phase. This behavior is consistent with the higher exploration rate, which increases the probability of exploratory actions and therefore introduces additional variability in the reward signal.

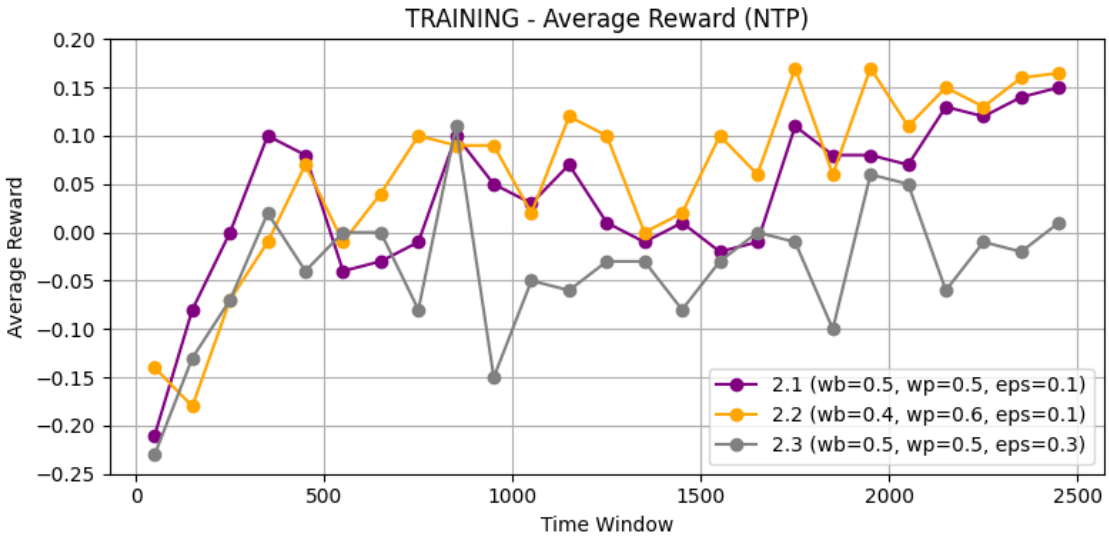

Figure 7.4: Training reward under different parameter configurations for the NTP attack scenario, shown as the average reward computed over intervals of 100 windows.

Similarly to what was observed in the DNS experiments, scenario 2.1 converges to a slightly lower average reward compared to scenario 2.2, as shown in Figure 7.4.

This difference can be attributed to the service degradation component included in the reward formulation. During training, the service degradation threshold is reached more frequently in scenario 2.1, activating the corresponding penalty term more often. As a consequence, the agent encounters a higher number of penalized states, which slightly lowers the overall average reward despite the correct mitigation behavior learned by the policy.

Figure 7.5 shows a representative segment of the test phase for scenario 2.1, illustrating the effect of the service degradation component on the agent's decisions.

Case 1 (azure circle). Around time window 418, the service degradation metric reaches the predefined threshold during the high plateau phase of a cyclic attack. In this situation the agent chooses the `ALLOW` action despite the ongoing *HIGH*-intensity attack. As a consequence, the Asymmetry Index increases, which can be observed in the upper plot where the AI value rises in the subsequent window. This increase leads the agent to switch back to the `DROP` action in the following window. However, after time window 420 the service degradation metric again reaches the threshold. As a result, when the attack intensity decreases to *LOW* in the next window, the agent allows the traffic in order to prevent the service degradation metric from exceeding the threshold.

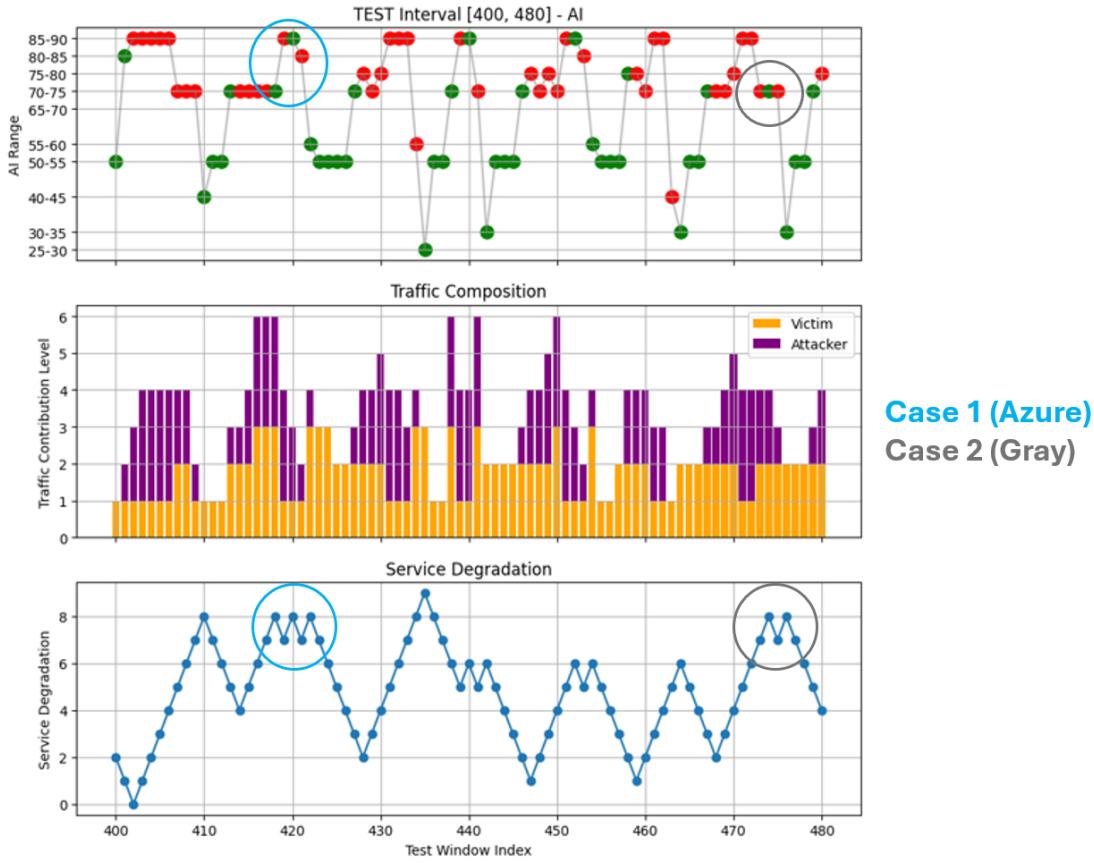

Figure 7.5: Example of agent decisions during the test interval for scenario 2.1, the top plot reports the Asymmetry Index observed by the agent and the corresponding action taken. The middle plot shows the traffic composition in each window, indicating the contribution levels of the victim and the attacker. The bottom plot reports the resulting service degradation experienced by the victim.

Case 2 (gray circle). A similar situation occurs around time window 474. In this case, the agent selects the `ALLOW` action while the attacker is generating *MEDIUM*-intensity traffic in order to preserve the service degradation metric within the acceptable bound. This decision allows the traffic generated in the subsequent window, where the attack intensity drops to *LOW*, to reach the victim without triggering additional mitigation actions.

The detection results reported in Figure 7.6 provide further insights into the effectiveness of the learned policies. For SILENT windows, the agent correctly selects the `ALLOW` action in approximately 89–91% of the cases across all scenarios, confirming that the system maintains a low rate of unnecessary blocking for legitimate traffic.

Under attack conditions, the detection performance again improves with the attack intensity. For MEDIUM and HIGH attacker profiles, the detection rate consistently exceeds 70% across all configurations, reaching values above 80% in the best case. Scenario 2.1 achieves the highest detection rate for MEDIUM attacks (82.95%), while maintaining strong performance for HIGH attacks (79.75%).

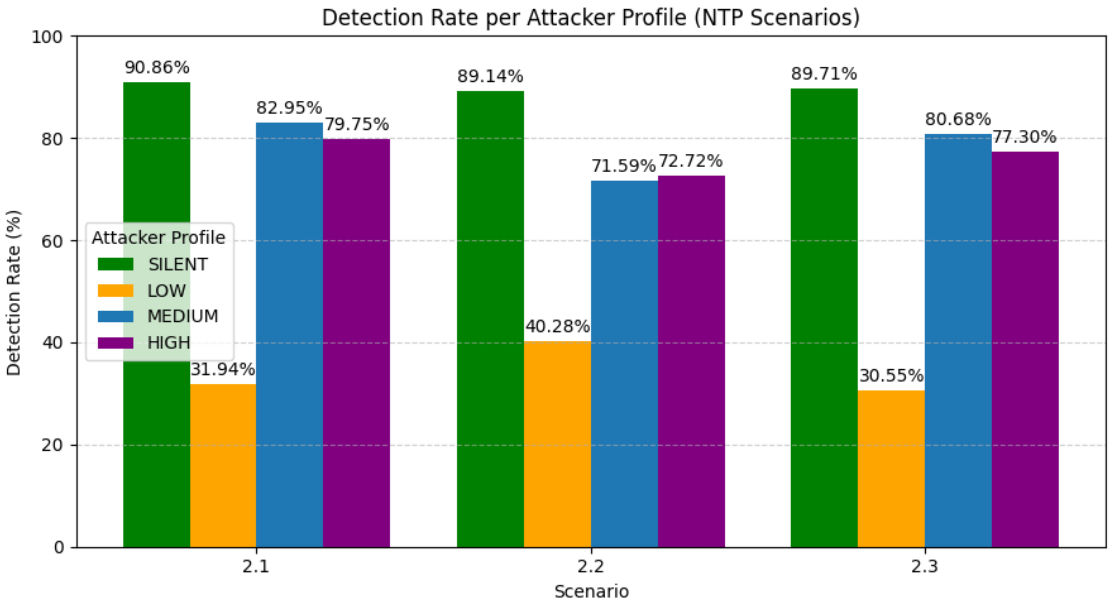

Figure 7.6: Detection rate of the proposed system across the NTP attack scenarios (2.1–2.3), reported for each attacker activity profile (SILENT, LOW, MEDIUM, HIGH). The results show how the detection performance varies with the intensity of the attack traffic.

A key difference compared to the DNS experiments emerges when analyzing the impact of the weighting coefficients used in the Asymmetry Index. In NTP amplification attacks, the amplification effect is reflected both in the number and size of packets generated by the reflector servers. In particular, when an NTP server receives a *monlist* request, it produces multiple response datagrams of approximately 440 bytes, while standard NTP request and response packets have a fixed size of 48 bytes, resulting in both a significant increase in the number of packets and a noticeable increase in the transmitted byte volume.

This characteristic explains the behavior observed in the experimental results. Scenario 2.1 and 2.3 use the same balanced weights and achieves strong detection performance. Conversely, scenario 2.2, which assing an higher weight to packet-level asymmetry, exhibits slightly lower detection rates for MEDIUM and HIGH attack intensities. This suggests that, in the NTP case, both packet-level and byte-level asymmetry contribute to the discrimination of attack traffic.

Finally, as observed in the DNS scenarios, the configuration with the higher exploration rate ($\epsilon = 0.3$) does not lead to significant improvements in detection performance.

7.3.3 Comparison with Port-Based Filtering

To provide a baseline comparison for the proposed reinforcement learning approach, we evaluate its performance against a simple threshold-based filtering strategy inspired by traditional port-based filtering mechanisms commonly adopted in firewall systems. In these approaches, mitigation decisions are triggered when a monitored traffic metric exceeds a predefined threshold, indicating potentially abnormal activity.

In our experimental setup, we adapt this concept to the programmable data plane context by using the Asymmetry Index as the monitored metric. Instead of learning a mitigation policy, the switch applies a static rule based on a fixed threshold. Specifically, when the AI observed within a time window exceeds the predefined threshold, the traffic is considered suspicious and mitigation is activated.

For this comparison, we use the best-performing parameter configuration identified for the NTP attack scenario in the RL experiments (2.1). The threshold-based policy is defined as follows: if the AI value computed for a given window satisfies $AI \geq \beta_H = 0.65$, the switch drops all incoming traffic associated with the monitored flow during the subsequent window. Otherwise, the traffic is allowed.

This rule emulates a simplified anomaly-based filtering mechanism where mitigation decisions rely solely on the instantaneous value of the traffic asymmetry metric, without considering historical context or adaptive learning. The comparison with this baseline allows us to assess the potential benefits of the RL-based policy in terms of its ability to adapt mitigation decisions to different traffic conditions and attack intensities.

To ensure a fair comparison between the two approaches, the same traffic traces used during the RL evaluation are reused for the threshold-based policy. The traffic is generated following the model described earlier, where each time window is assigned a traffic intensity level sampled pseudo-randomly according to the predefined traffic profiles for both the victim and the attacker. To ensure reproducibility and identical traffic conditions, the sequence of windows is generated using the same seed. The reported results are obtained by evaluating the mitigation decisions over a test phase consisting of 500 time windows.

Figure 7.7 reports the distribution of attacker activity levels observed in the subsequent time window conditioned on the mitigation decision taken in the current window. This analysis provides insight into the predictive capability of the

mitigation policy by showing how often `DROP` and `ALLOW` decisions precede different attack intensities.

Since the attack intensity of each window is generated by the traffic model, the ground truth corresponds to the attacker activity level (SILENT, LOW, MEDIUM, HIGH) associated with the subsequent window.

A desirable mitigation behavior is characterized by DROP decisions that are frequently followed by MEDIUM or HIGH attack windows, while ALLOW decisions should ideally precede SILENT traffic periods. In particular, an adaptive mitigation policy should be able to intervene during the early or intermediate phases of an attack, rather than reacting only when the traffic asymmetry becomes clearly dominant.

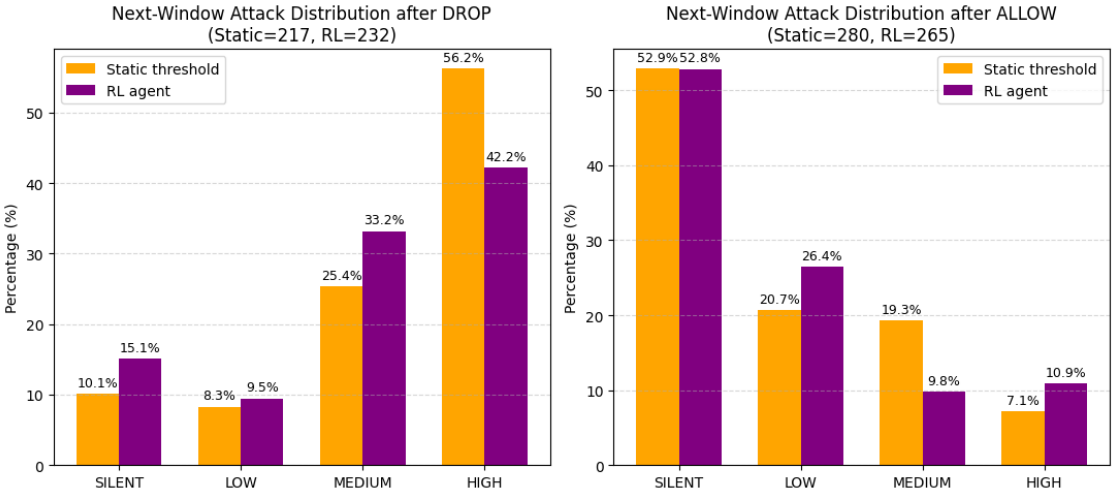

Figure 7.7: Distribution of attacker activity levels observed in the subsequent time window conditioned on the mitigation decision taken in the current window, comparing the RL-based agent and a static threshold policy in the NTP attack scenario 2.1.

Considering `DROP` decisions, the static threshold policy is strongly associated with high-intensity attack windows, with approximately 56% of the subsequent windows corresponding to the *HIGH* attacker profile. This behavior reflects the nature of the threshold mechanism, which triggers mitigation only when the asymmetry index clearly exceeds the predefined limit. In contrast, the RL agent exhibits a more balanced distribution across attack intensities: while the proportion of *HIGH* windows following a `DROP` decision decreases to about 42%, the percentage of *MEDIUM* windows increases to approximately 33%. This indicates that the RL policy tends to intervene not only during peak attack phases but also during intermediate stages of the attack evolution, allowing earlier mitigation compared to the purely threshold-based strategy.

For `ALLOW` decisions, both approaches are predominantly followed by *SILENT* windows (around 53%), indicating that benign traffic is frequently preserved. However, the RL agent allows a higher proportion of LOW attack windows (26% compared to 21% for the static threshold). This behavior can be partly explained by

the service degradation dynamics discussed earlier. Since frequent mitigation actions may negatively impact legitimate traffic, the agent learns to tolerate some low-intensity attack activity in order to avoid unnecessary filtering and maintain service availability.

At the same time, the RL policy reduces the proportion of *MEDIUM* attack windows that are allowed. Unlike the static threshold approach, which relies exclusively on the instantaneous value of the asymmetry index, the RL agent incorporates historical information into the decision process. By exploiting the temporal context of previous observations and actions, the agent can identify potentially harmful situations even when the current AI value is not strictly above the alarm threshold, allowing it to interpret borderline traffic conditions as part of an evolving attack phase.

The right-hand plot of Figure 7.7 highlights another intrinsic limitation of threshold-based mitigation strategies: HIGH-intensity windows may still be preceded by `ALLOW` decisions when earlier *MEDIUM* attack phases do not dominate the asymmetry index due to concurrent high victim traffic. In these situations, the static policy reacts only once the attack becomes clearly dominant.

This limitation observed is also consistent with behaviors commonly reported in real-world traffic monitoring systems. In practice, mitigation mechanisms based on fixed thresholds often rely on volumetric indicators such as traffic asymmetry or flow rate anomalies. In such cases, moderate attack activity may remain undetected when it occurs concurrently with high legitimate traffic levels, a phenomenon commonly referred to as traffic masking.

A related effect can be observed in the proportion of *SILENT* windows following `DROP` decisions, mitigation may continue even when the attack intensity is already decreasing, leading to filtering actions during benign traffic periods near the end of the attack.

Despite the relatively limited number of training windows and the simplified representation of the system state obtained through feature discretization, the RL-based agent achieves promising results. Although the learned policy is not yet optimal, the results demonstrate the potential of the proposed approach to learn more flexible mitigation strategies compared to static rules. In particular, the agent exploits historical traffic observations to adapt its mitigation behavior to the evolving attack dynamics.

Overall, these findings indicate that the RL-based approach is particularly beneficial in traffic conditions where the instantaneous value of the asymmetry index alone is insufficient to clearly identify malicious activity. By leveraging temporal information, the RL agent can anticipate evolving attack phases and adapt mitigation decisions accordingly, whereas static threshold-based strategies remain inherently reactive.

Chapter 8

Conclusion and Future Works

8.1 Limitations and Threats to Validity

Despite the encouraging results obtained in the experimental evaluation, several limitations of the current implementation should be acknowledged.

First, the discretization intervals used for the monitored metrics were calibrated according to the practical traffic generation limits of the experimental environment. The prototype was deployed on a software switch running inside a virtual machine on a commodity laptop, which inherently constrains the maximum sustainable packet rate and traffic volume. To ensure stable execution and avoid system-level bottlenecks that could bias the experimental results, the discretization bins were therefore defined based on the maximum traffic intensity that the testbed can reliably sustain. Consequently, while the discretization strategy itself is conceptually scalable, the specific bin configuration adopted in this work reflects the computational constraints of the experimental platform rather than intrinsic limitations of the proposed method. For similar reasons, the PAF and BAF values used in the traffic generator are slightly lower than those reported in Rossow's empirical study, as reproducing the full amplification levels observed in real-world attacks would exceed the capabilities of the experimental setup.

A second limitation concerns the reward formulation and the associated reinforcement learning dynamics. In this work the reward function is defined using a discrete set of values ($r \in \{-1,0,1\}$). While this simplified formulation allows the learning process to be efficiently implemented in the data plane, it also reduces the sensitivity of the update rule to variations in the learning rate and discount factor. In particular, since arithmetic operations are implemented through discretized lookup tables, intermediate results are subject to rounding effects. As a consequence, moderate changes in the learning rate (α) and discount factor (γ) have only a limited impact on the numerical update of the Q-values. Exploring more expressive reward formulations was not investigated within the scope of this work due to time constraints, but represents an interesting direction for future research.

Another potential threat to validity concerns the duration of the training phase. The agent was trained for 2500 time windows, which represents a practical compromise between learning effectiveness and experimental feasibility. As discussed in the

experimental setup, the adopted traffic model yields approximately 3000 reachable states, although many of them occur rarely under the considered traffic patterns. Moreover, since each time window lasts 10 seconds, substantially increasing the training duration would lead to very long experimental runs.

The definition of the service degradation component in the reward function also represents a design choice that may influence the observed behavior of the agent. In particular, the threshold used to represent unacceptable service degradation was selected empirically and therefore remains somewhat arbitrary. The purpose of this component was to investigate whether the agent could learn to balance security and service quality by avoiding excessive packet drops that could harm legitimate traffic. As illustrated by several snapshots (Figure 7.3 and 7.5) analyzed during the evaluation, the agent occasionally decides to allow traffic even during *MEDIUM* or *HIGH* attack windows in order to prevent the service degradation metric from exceeding the predefined threshold. While this behavior highlights the intended trade-off between mitigation and service availability, different threshold values could lead to different policy behaviors.

8.2 Future Work

Several directions can be explored to extend and improve the approach presented in this work.

First, future work could investigate more expressive reward formulations, such as larger-range reward values (e.g., $r \in [0,100]$), which could provide richer feedback to the agent and potentially enable faster and more accurate convergence of the Q-values.

Currently, the agent operates at the granularity of individual flows. While this design allows localized mitigation decisions, it does not explicitly exploit correlations between different flows targeting the same service. Future research could investigate mechanisms to aggregate the knowledge learned across multiple reflector servers associated with the same protocol. Such aggregation strategies could enable the system to better capture protocol-level attack patterns and improve the effectiveness of mitigation decisions.

Another promising direction concerns the deployment of the proposed approach on programmable hardware switches rather than on a software-based testbed. Running the system on real hardware would allow experiments with significantly higher traffic rates and more complex traffic patterns, including large-scale amplification attacks generated by networks composed of hundreds of distributed nodes. Such a setup would better approximate real-world attack scenarios and enable the evaluation of the mitigation mechanism under more realistic conditions. Moreover, hardware deployment would make it possible to reduce the duration of the observation windows, allowing longer training processes and a finer-grained observation of traffic dynamics.

Furthermore, the current implementation relies on a binary action space in which the agent decides whether to `ALLOW` or `DROP` traffic. Future work could explore more flexible mitigation strategies by introducing a continuous action space,

enabling the agent to selectively throttle or partially drop traffic rather than applying a purely binary decision. In addition, the current state representation only considers a short temporal history composed of two consecutive time windows. While this design keeps the state space manageable for the data plane implementation, extending the observation history could allow the agent to capture longer-term traffic dynamics and potentially improve the detection of more complex attack patterns. Investigating the impact of larger temporal histories therefore represents another interesting direction for future work.

Finally, the approach presented in this thesis focuses on amplification-based attacks exploiting protocols such as DNS and NTP. Extending the methodology to other categories of denial-of-service attacks, and ultimately developing a unified reinforcement learning framework capable of mitigating multiple DoS attack vectors, represents an important direction for future research.

8.3 Work Overview

Distributed Denial of Service attacks continue to represent a major threat to Internet availability, with amplification-based techniques being particularly effective due to their ability to generate large traffic volumes by exploiting protocol asymmetries. In this context, mitigating malicious traffic as early as possible in the network becomes essential. Programmable data planes provide a promising opportunity, as they enable monitoring and mitigation logic to be executed directly within network switches at line rate.

Despite these capabilities, most existing approaches either rely on static filtering rules or use machine learning techniques trained offline and executed outside the forwarding path. While reinforcement learning has shown potential for adaptive network defense, current solutions rarely operate entirely within the programmable data plane and often focus primarily on flooding-based attacks rather than amplification scenarios.

In this thesis, we investigated the feasibility of performing reinforcement learning directly in the programmable data plane for the mitigation of amplification-based DDoS attacks. The mitigation problem was modeled as a Markov Decision Process in which the programmable switch acts as a learning agent observing traffic asymmetry patterns and selecting mitigation actions. A Q-learning algorithm was implemented directly in the P4 data plane, allowing the agent to learn mitigation policies online without relying on external datasets or control-plane intervention.

The proposed system introduces an amplification-aware traffic representation based on asymmetry metrics that can be computed at line rate, and integrates these features with a reward formulation designed to balance attack mitigation and service availability. A complete prototype was implemented and evaluated in a controlled experimental environment using DNS and NTP amplification attack scenarios.

The experimental results show that the agent is able to learn effective mitigation policies and adapt its behavior to different attack intensities. In particular, the RL-based approach achieves high detection rates for medium and high attack

phases while maintaining a low rate of unnecessary filtering during benign traffic conditions. Compared to a static threshold-based filtering policy, the learned policy demonstrates greater flexibility by leveraging temporal information from previous observations, enabling more context-aware mitigation decisions.

Although the current implementation relies on discretized features and a software-based experimental environment, the results provide a proof of concept that reinforcement learning can be successfully embedded within programmable switches. Future work could investigate richer reward formulations, extended state representations, and deployments on programmable hardware switches to evaluate the approach under more realistic traffic conditions.

Overall, this work shows that combining programmable data plane technologies with reinforcement learning represents a promising direction toward adaptive, in-network DDoS mitigation mechanisms capable of reacting to evolving attack dynamics directly within the network infrastructure.

Bibliography

[1] C. Douligeris and A. Mitrokotsa, "Ddos attacks and defense mechanisms: classification and state-of-the-art," *Comput. Networks*, vol. 44, no. 5, pp. 643–666, 2004. [Online]. Available: https://doi.org/10.1016/j.comnet.2003.10.003

[2] C. Zheng, X. Hong, D. Ding, S. Vargaftik, Y. Ben-Itzhak, and N. Zilberman, "In-network machine learning using programmable network devices: A survey," *IEEE Commun. Surv. Tutorials*, vol. 26, no. 2, pp. 1171–1200, 2024. [Online]. Available: https://doi.org/10.1109/COMST.2023.3344351

[3] "Targeted by 20.5 million ddos attacks, up 358% year-over-year: Cloudflare's 2025 q1 ddos threat report," https://blog.cloudflare.com/ddos-threat-report-for-2025-q1/.

[4] "Hyper-volumetric ddos attacks skyrocket: Cloudflare's 2025 q2 ddos threat report," https://blog.cloudflare.com/ddos-threat-report-for-2025-q2/.

[5] "Digital aftershocks: Collateral damage from ddos attacks – netscout 1h 2025 ddos threat intelligence report," https://www.netscout.com/resources/threat-report/ddos-threat-intelligence-report-digital-aftershocks.

[6] "Massive ddos attack delivered 37.4tb in 45 seconds, equivalent to 10,000 hd movies, to one victim ip address — cloudflare blocks largest cyber assault ever recorded," https://www.tomshardware.com/tech-industry/cyber-security/massive-ddos-attack-delivered-37-4tb-in-45-seconds-equivalent-to-10-000-hd-movies-to-one-victim-ip-address-cloudflare-blocks-largest-cyber-assault-ever-recorded.

[7] ENISA, "Enisa threat landscape 2024," https://www.enisa.europa.eu/publications/enisa-threat-landscape-2024, ENISA, Tech. Rep., 2024.

[8] "Eleven11bot botnet grows to over 86,000 devices," https://www.cyber.nj.gov/Home/Components/News/News/1646/214?utm_source=chatgpt.com.

[9] "Global operation targets noname057(16) pro-russian cybercrime network," https://www.europol.europa.eu/media-press/newsroom/news/global-operation-targets-noname05716-pro-russian-cybercrime-network.

[10] B. Guttman and E. A. Roback, "An introduction to information security," *NIST Special Publication*, vol. 800-12r1, pp. 88–90, 2017. [Online]. Available: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-12r1.pdf

[11] S. Ismail, H. R. Hassen, M. Just, and H. Zantout, "A review of amplification-based distributed denial of service attacks and their mitigation," *Comput. Secur.*, vol. 109, p. 102380, 2021. [Online]. Available: https://doi.org/10.1016/j.cose.2021.102380

[12] C. Rossow, "Amplification hell: Revisiting network protocols for ddos abuse," in *21st Annual Network and Distributed System Security Symposium, NDSS*

2014, San Diego, California, USA, February 23-26, 2014. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/amplification-hell-revisiting-network-protocols-ddos-abuse

[13] Y. Chen, S. Layeghy, L. D. Manocchio, and M. Portmann, "P4-NIDS: high-performance network monitoring and intrusion detection in P4," *CoRR*, vol. abs/2411.17987, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2411.17987

[14] T. Mai, S. Garg, H. Yao, J. Nie, G. Kaddoum, and Z. Xiong, "In-network intelligence control: Toward a self-driving networking architecture," *IEEE Netw.*, vol. 35, no. 2, pp. 53–59, 2021. [Online]. Available: https://doi.org/10.1109/MNET.011.2000412

[15] E. Paolini, L. D. Marinis, D. Scano, and F. Paolucci, "In-line any-depth deep neural networks using P4 switches," *IEEE Open J. Commun. Soc.*, vol. 5, pp. 3556–3567, 2024. [Online]. Available: https://doi.org/10.1109/OJCOMS.2024.3411071

[16] W. Liu, C. Liang, Y. Cui, J. Cai, and J. Luo, "Programmable data plane intelligence: Advances, opportunities, and challenges," *IEEE Netw.*, vol. 37, no. 5, pp. 122–128, 2023. [Online]. Available: https://doi.org/10.1109/MNET.124.2200113

[17] C. Zheng, B. Rienecker, and N. Zilberman, "QCMP: load balancing via in-network reinforcement learning," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Future of Internet Routing & Addressing, FIRA@SIGCOMM 2023, New York, NY, USA, 10 September 2023*. ACM, 2023, pp. 35–40. [Online]. Available: https://doi.org/10.1145/3607504.3609291

[18] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction, 2nd Edition*. MIT Press, 2018. [Online]. Available: http://www.incompleteideas.net/book/the-book-2nd.html

[19] S. Khozam, G. Blanc, S. Tixeuil, and E. Totel, "Ddos mitigation while preserving qos: A deep reinforcement learning-based approach," in *10th IEEE International Conference on Network Softwarization, NetSoft 2024, Saint Louis, MO, USA, June 24-28, 2024*. IEEE, 2024, pp. 369–374. [Online]. Available: https://doi.org/10.1109/NetSoft60951.2024.10588889

[20] K. A. Simpson, S. Rogers, and D. P. Pezaros, "Per-host ddos mitigation by direct-control reinforcement learning," *IEEE Trans. Netw. Serv. Manag.*, vol. 17, no. 1, pp. 103–117, 2020. [Online]. Available: https://doi.org/10.1109/TNSM.2019.2960202

[21] A. Ganesan and K. Saraç, "In-network reinforcement learning for attack mitigation using programmable data plane in SDN," in *33rd International Conference on Computer Communications and Networks, ICCCN 2024, Kailua-Kona, HI, USA, July 29-31, 2024*. IEEE, 2024, pp. 1–9. [Online]. Available: https://doi.org/10.1109/ICCCN61486.2024.10637621

[22] Y. Zhang and Y. Cheng, "An amplification ddos attack defence mechanism using reinforcement learning," in *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation, SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI 2019, Leicester, United Kingdom, August 19-23, 2019*. IEEE, 2019, pp. 634–639. [Online]. Available: https:

//doi.org/10.1109/SmartWorld-UIC-ATC-SCALCOM-IOP-SCI.2019.00145

[23] W. K. Kochenderfer M., Wheeler T., "Algorithms for decision making," *MIT Press*, 2022.

[24] P4 Language Consortium, "simple_switch: Behavioral model v2 documentation," https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md, 2024, accessed: January 2026.

[25] S. Sampognaro, "In-switch rl ddos defense: Reinforcement learning for amplification-based ddos mitigation in programmable data planes," https://github.com/SimoneSampognaro/in-switch-rl-ddos-defense, 2026, gitHub repository.

[26] L. Feinstein, D. Schnackenberg, R. Balupari, and D. Kindred, "Statistical approaches to ddos attack detection and response," in *Proceedings DARPA Information Survivability Conference and Exposition*, vol. 1, 2003, pp. 303–314 vol.1.

[27] P4 Language Consortium, "P4 tutorials," https://github.com/p4lang/tutorials, 2026, gitHub repository.

[28] A. Kuzmanovic and E. W. Knightly, "Low-rate tcp-targeted denial of service attacks and counter strategies," *IEEE/ACM Trans. Netw.*, vol. 14, no. 4, pp. 683–696, 2006. [Online]. Available: http://doi.acm.org/10.1145/1217648

[29] M. Antonakakis, T. April, M. D. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou, "Understanding the mirai botnet," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, E. Kirda and T. Ristenpart, Eds. USENIX Association, 2017, pp. 1093–1110. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis

[30] P. Foremski *et al.*, "Dns observatory: The big picture of the dns," in *Internet Measurement Conference (IMC)*, 2019.