Politecnico di Torino

Master of Science in Cybersecurity

Master Degree Thesis

A Novel AI Based Algorithm for Automatic Reordering of Firewall Rules

Supervisors
prof. Fulvio Valenza
prof. Daniele Bringhenti
prof. Riccardo Sisto
dott. Gianmarco Bachiorrini

Candidate
Alessandro Mulassano

Academic Year 2024-2025

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Firewalls represent a fundamental component of network security, as they filter traffic based on predefined policies. Their operation relies on analyzing packets in transit and comparing them against an ordered set of rules. However, as the number of rules increases, processing speed tends to decline significantly, with direct repercussions on the network's overall performance.

In addition to this performance aspect, there is the complexity associated with configuring security policies [1]. In corporate settings, firewall rules are often the result of years of modifications, layering, and maintenance carried out by different administrators. This not always coordinated evolution can give rise to anomalies in filtering rules. Several studies have addressed this issue [2] [3]: the former proposes a systematic and structured approach to identify and correct anomalies, whereas the latter adopts a semi-automatic methodology that supports administrators in the most complex phases of resolution.

In parallel, other research has explored the possibility of automating firewall configuration, thereby reducing manual intervention and making systems more adaptive. For example, [4] analyses the challenges involved in managing distributed firewalls in virtualized environments, where complexity renders traditional methods poorly scalable. The work in [5] introduces optimization criteria oriented towards energy sustainability, while [6] and [7] focus on transition management and update scheduling, with the aim of reducing the impact of frequent reconfigurations. Finally, [8] proposes an autonomous model capable of reacting dynamically to potential attacks by modifying rules without human intervention.

This work focuses on the specific aspect of *policy reordering*; in this context the *Optimal Rule Ordering Problem* [9] is one of the main challenges, as it requires determining a sequence for rule evaluation that minimizes processing time and the number of incorrectly handled packets.

Moreover it's possible to divide the optimization in two broad approaches: the **static** case, in which the order of rules is defined once and remains unchanged unless manually modified by an administrator, making it poorly flexible with respect to evolving traffic; by contrast the **dynamic** case updates the sequence periodically or in real time, prioritizing the most frequently used rules and thereby reducing processing time.

In this thesis, we develop a **dynamic** and adaptive **reordering** mechanism, implemented within a **smart firewall** capable of improving operational efficiency and reducing misses. To this end, a **Reinforcement Learning** agent is adopted—an approach particularly well-suited to variable contexts, in which the optimal strategy is learned through continuous interaction with the environment.

In particular, we employ the **Q-learning** algorithm, which makes it possible to learn an effective reordering policy for a packet filtering firewall, ensuring rule correctness and optimizing overall performance. The reference context is that of a packet filtering firewall, in which each rule is defined by 5 different fields: `sourceIP`, `destinationIP`, `sourcePort`, `destinationPort`, `protocol`. The analysis focuses on *sub-optimization* anomalies, excluding conflicting ones, and aims to demonstrate how an artificial intelligence–based approach can significantly improve policy management.

1.1 Thesis Structure

The organization of the thesis is as follows:

- **Chapter 2 - Firewalls:** We introduce firewalls and their role in network defense, outlining core concepts such as packet filtering and access control. The chapter surveys the main families (stateless, stateful, application-aware) and clarifies first-matching semantics and how rule ordering impacts latency.
- **Chapter 3 - Artificial Intelligence and Machine Learning:** We provide a concise background on AI/ML, then focus on reinforcement learning: agents, environments, rewards, and exploration–exploitation. Particular attention is given to tabular Q-learning, which forms the basis for the agents used in this work.
- **Chapter 4 - Thesis Objectives:** We define the problem and goals of the study, detailing what we aim to optimize and why.
- **Chapter 5 - Approach and Methodology:** We present the overall approach to the rule reordering problem and the proposed architectures. This includes the smart firewall (stateless and state-based variants) and the digital twin setup that couples a learning agent with a traditional firewall, along with the experimental workflow.
- **Chapter 6 - Implementation and Validation:** We describe how each agent and the traditional component are implemented, highlighting the key design choices and interfaces. The chapter then reports the validation procedure and results, including scalability across different rule set sizes and update intervals.
- **Chapter 7 - Conclusions:** We summarize the main findings and their implications for AI assisted firewalling. The chapter closes with avenues for future work, such as dynamic state abstraction, drift-aware operation, larger-scale validation, and distributed deployment.

Chapter 2

Firewalls

A firewall is a network device that functions as a filter between the protected network and the outside world. Every inbound and outbound packet must traverse this control point, where it is evaluated against the local security policy: only permitted communications are allowed to pass, while all others are denied. The device is engineered to resist compromise, thereby preserving the integrity of the protected perimeter [10].

2.1 Overall Functioning

The core of a firewall is its *security policy*, an ordered sequence of filtering rules that prescribes how to handle every ingress and egress packet. Each IP packet is processed as follows [11]:

1. **Packet inspection**: the packet is parsed and evaluated against the entire rule set. can be a single or a range of values and represents the possible values of the corresponding field in actual packets which match this rule.

2. **Conditions** : set of attributes that represents the possible values such as source and destination IP addresses, TCP/UDP ports, the transport protocol.

3. **Actions** : if the conditions of a rule are satisfied, one of the following action is applied:

   - `ALLOW` (or `ACCEPT`): the packet is forwarded to its destination.
   - `DENY` (or `DROP`): the packet is blocked and discarded.

4. **Default action**: if a packet doesn't match any rule, a fallback action is taken, typically `DENY` for security, although `ALLOW` may be used in specific scenarios.

To make the notion of a policy more concrete, Figure 2.1 presents a representative example of a firewall *rule table*, showing how conditions and actions are arranged in sequence to manage network traffic.

order	protocol	src_ip	src_port	dst_ip	dst_port	action
1:	tcp,	140.192.37.20,	any,	*.*.*.*,	80,	deny
2:	tcp,	140.192.37.*,	any,	*.*.*.*,	80,	accept
3:	tcp,	*.*.*.*,	any,	140.192.37.40,	80,	accept
4:	tcp,	140.192.37.*,	any,	140.192.37.40,	80,	deny
5:	tcp,	140.192.37.30,	any,	*.*.*.*,	21,	deny
6:	tcp,	140.192.37.*,	any,	*.*.*.*,	21,	accept
7:	tcp,	140.192.37.*,	any,	140.192.37.40,	21,	accept
8:	tcp,	*.*.*.*,	any,	140.192.37.40,	21,	accept
9:	tcp,	*.*.*.*,	any,	*.*.*.*,	any,	deny
10:	udp,	140.192.37.*,	any,	*.*.*.*,	53,	accept
11:	udp,	*.*.*.*,	any,	140.192.37.*,	53,	accept
12:	udp,	*.*.*.*,	any,	*.*.*.*,	any,	deny

Figure 2.1. Example of a firewall *rule table*

2.1.1 Rule Resolution Strategies

When a packet satisfies multiple rules simultaneously, the firewall applies a resolution strategy to decide which action to impose among the potentially conflicting ones [12]:

- **First Matching Rule (FMR)**: examines the rules consecutively and ceases at the initial match, executing the specified action of that rule.
- **Allow Takes Precedence (ATP)**: when there is an ALLOW and DENY conflict, the ALLOW prevails, reducing the risk of false positives.
- **Deny Takes Precedence (DTP)**: in the presence of conflicts, DENY always prevails, favoring a conservative security posture.
- **Most Specific Takes Precedence (MSTP)**: between all applicable rules, selects the most specific one.
- **Least Specific Takes Precedence (LSTP)**: between all applicable rules, selects the least specific one.

2.1.2 Denylist vs Allowlist

Denylist (Blacklist)

In a denylist (blacklist) configuration, the policy explicitly enumerates traffic that must be blocked. Each rule encodes conditions that, when satisfied, result in a `DENY` decision for the packet. In practice, the firewall evaluates incoming packets against the denylist and discards any that match; non matching traffic is forwarded under a default allow posture. This model performs well when malicious patterns are well delineated and relatively stable, but it requires regular maintenance to keep pace with newly discovered threats and vulnerabilities.

Allowlist (Whitelist)

An allowlist (or whitelist) adopts a more restrictive posture: only traffic that satisfies narrowly defined criteria is permitted, with all other traffic denied by default. The firewall checks whether a packet matches an allowlist entry and forwards it if so; any packet that does not match is automatically blocked, consistent with a default-deny policy. While this strategy offers a high level of security by substantially reducing the risk of unauthorized access, it entails greater administrative overhead, as legitimate communications must be specified in advance and the rules updated promptly as requirements evolve.

2.2 Classification of Firewalls

Firewalls can be implemented as dedicated hardware appliances, as software solutions, or in hybrid configurations that combine both. The goal is to provide the highest possible level of protection and traffic control without unduly compromising performance or forwarding throughput. Depending on security requirements and the desired degree of visibility into traffic flows, several types of firewalls can be distinguished [13] [14].

2.2.1 Packet Filtering Firewall

The *packet filtering firewall* (Figure 2.2) (also known as a Layer-3 network firewall) is the most common and straightforward form of filtering. It inspects each packet in isolation, comparing its headers against a predefined rule set. If a packet satisfies the specified criteria, it is forwarded; otherwise, it is dropped.

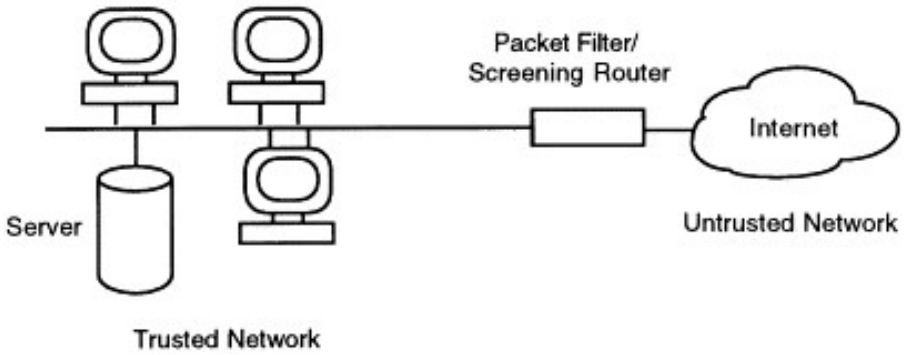


Figure 2.2. Example of a *packet filtering firewall*

Typical matching criteria include:

- protocol type (IP, TCP, UDP, ICMP, etc.);
- source and destination IP addresses, for example, permitting only traffic from 192.168.1.1 to 192.168.1.100;

- source and destination ports, for instance, accepting only TCP packets destined for port 25 (SMTP) and rejecting those for port 21 (FTP);
- traffic direction (ingress or egress).

Advantages

- It has a simple configuration.
- It is trasparent for the users, who perceive no change to the data flow.
- It has an high speed and low latency due to minimal checks.

Disadvantages

- It doesn't have authentication or payload inspection, it's unable to distinguish harmful content encapsulated in otherwise legitimate packets.
- It is vulnerable to various attacks, such as denial of service (DOS), IP spoofing, and tiny fragment attacks, because it relies on simple header matching.

2.2.2 Circuit Level Gateway

A *circuit level gateway* (Figure 2.3) operates at the session layer (OSI layer 5). It monitors the TCP handshake (and the corresponding state for UDP) between the local and remote host to determine whether the session being opened is legitimate. Unlike packet filters, it does not inspect the content of individual packets; instead, it focuses on connection establishment and termination.

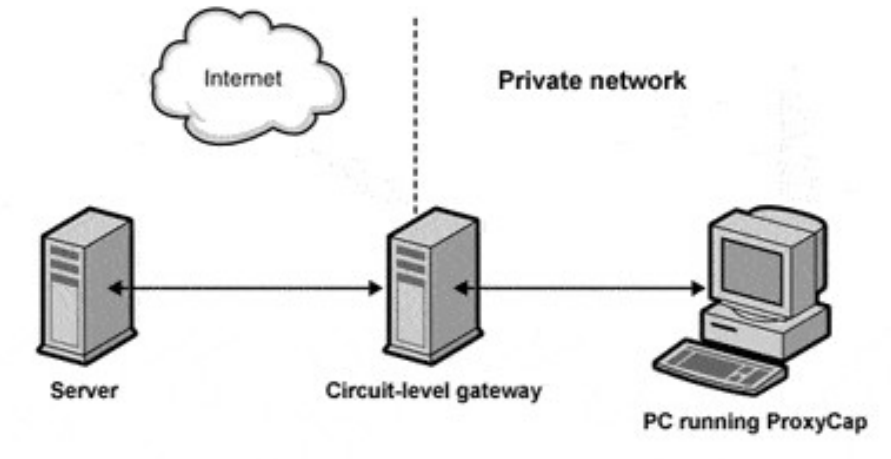


Figure 2.3. Example of a *circuit level gateway*

When an internal host starts an outbound connection, the gateway checks that the characteristics of the handshake (sequence numbers, the ports in use, TCP

flags) comply with the configured security policies. If the check succeeds, a "virtual circuit" is established and subsequent packets are allowed to flow without further inspection until the session closes.

Advantages

- It provides centralized oversight of active sessions and, compared with stateless filters, narrows the attack surface.
- Because it avoids parsing the payload, its processing burden is lower than that of proxies operating at the application layer.

Disadvantages

- It doesn't inspect the packets, so there is no visibility of the application content, and so cannot block malicious payloads embedded in otherwise legitimate flows.
- After a session is established, packets are no longer inspected one by one, which creates the possibility of abuse during the session.

2.2.3 Application Level Gateway

An *application level gateway* (Figure 2.4), also known as an application firewall or proxy, operates at OSI layer 7. It acts as a dedicated intermediary for each service (HTTP, FTP, SMTP, DNS, and so on). The internal client connects to the gateway, which authenticates the request, examines it, and filters it by analyzing the message content in detail (the payload). The gateway then opens a new connection to the destination server only if the communication complies with the security policy.

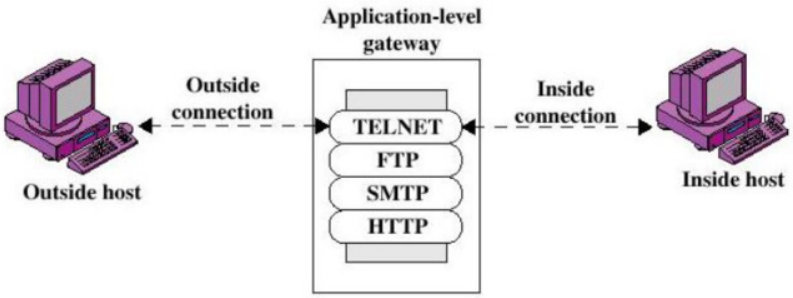


Figure 2.4. Example of an *application level gateway*

Advantages

- It hides internal hosts by exposing only the proxy address to external networks.
- It supports the integration of authentication and access control mechanisms.
- It enables payload inspection.
- It gives a detailed recording and logs of the connections.

Disadvantages

- It has higher latency due to the need to analyze and reconstruct traffic.
- It has also a complex configuration, often requiring protocol specific proxies.
- It can become a potential bottleneck due to high traffic environments without load balancing.

2.2.4 Stateful Inspection Firewall

A *stateful inspection firewall* (Figure 2.5) keeps a record of all active connections (flows) that traverse it. This allows the device to classify each packet as the start of a new connection, a part of an existing connection, or an invalid packet. When the first packet of a new flow arrives (for example, a TCP SYN), the firewall checks it against the configured rules and, if permitted, creates an entry in the state table. All subsequent packets that belong to that connection are then accepted automatically until the flow is closed.

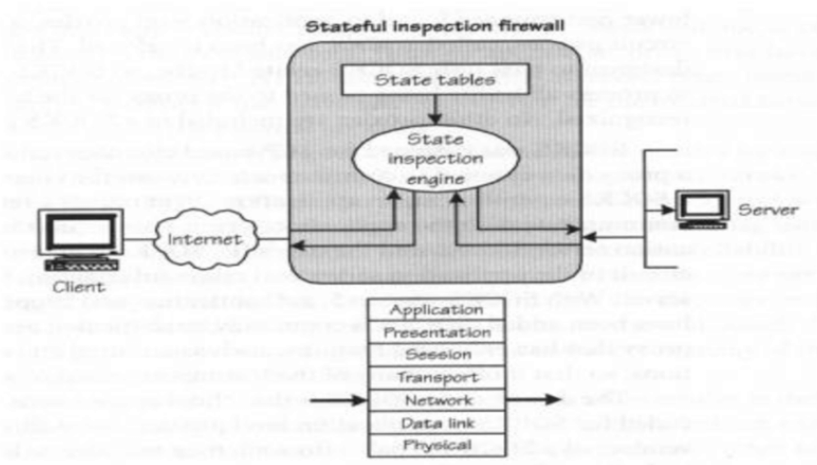


Figure 2.5. Example of a *stateful inspection firewall*

Advantages

- Precise control over connections, with the ability to distinguish legitimate traffic from malicious traffic.
- A sound balance between security and efficiency, since only the packets that open a connection are examined in detail.
- Added protection against spoofing and fragmentation as a result of state tracking.

Disadvantages

- Possible problems with slow or idle connections, which may be closed too early.
- Vulnerable to denial of service when the state table becomes saturated.

2.3 Anomalies in Firewall Policies

When defining a firewall rule set, policy anomalies may arise, especially when rules are authored or modified by multiple administrators at different times or when the number of rules is large. An anomaly occurs when the interaction among two or more rules leads to unintended behaviour or inefficiencies in network traffic handling.

In particular it's possible to distinguish:

- **Intra-policy anomalies**: anomalies among rules within the same firewall;
- **Inter-policy anomalies**: anomalies among rules belonging to different firewalls.

In this work we focus on **Intra-policy anomalies**, further divided into two main categories [12]:

1. **Conflict anomalies**: rules that, when applied together, yield conflicting actions or behaviours that depend on the evaluation order;
2. **Sub-optimization anomalies**: redundant or ineffective rules that degrade performance or the clarity of the configuration.

2.3.1 Conflict anomalies

Conflict anomalies arise when two rules interfere in a way that leads to contradictory behaviour. They generally cannot be resolved automatically without the risk of altering the semantics of the security policy, and therefore require manual intervention. Three main types are usually distinguished:

1. **Contradiction**
   Two rules match the same conditions but prescribe opposite actions (for example, one allows and the other denies the same traffic).

2. **Shadowing Conflict**
   A higher priority rule overrides another rule that targets the same traffic but prescribes a different action; consequently, the lower priority rule is never reached during evaluation.

3. **Correlation**
   Two rules partially overlap and specify different actions, so they apply different decisions to subsets of the traffic. Some packets are handled by both rules, others by only one of them.

2.3.2 Sub-optimization anomalies

Sub-optimization anomalies are inefficiencies in the firewall policy. They do not directly compromise security, but they lead to suboptimal use of resources and greater administrative complexity. They are mainly divided into four cases:

1. **Irrelevance**
   A rule is irrelevant if it can never be applied—for example, because the source or destination addresses do not belong to networks protected by the firewall. Removing it does not change behaviour, but it improves performance.

2. **Duplication**
   Two rules are duplicates if they match exactly the same conditions and specify the same action. In this case, the rule with lower priority can be removed without side effects.

3. **Shadowing Redundancy**
   A higher priority rule fully covers the same traffic and enforces the same decision; the lower priority rule is therefore unreachable and never takes effect.

4. **Unnecessary**
   A rule is unnecessary when the packets it would match are already covered by a preceding rule with the same action and there is no rule with a priority between them with different action that can match that packet.

2.4 From Background to the Present Work

In this work we consider a **firewall** that adopts a *FMR* (First Matching Rule) resolution strategy, with the goal of optimizing the rule order while preserving the coherence of the security policy. The firewall enforces an **allowlist**, that is, a set of rules that explicitly specifies permitted traffic and implicitly blocks all other flows.

For simplicity and clarity, we focus on a **packet filtering firewall**, the simplest model, which operates at the network and transport layers by examining packet headers. Rules and packets are represented as:

$$(\text{srcIP}, \text{dstIP}, \text{srcPort}, \text{dstPort}, \text{Protocol})$$

Finally, the rule set under study contains only **sub-optimization anomalies**; **conflict anomalies** are excluded. This choice allows us to concentrate the analysis and experiments on the automatic detection and removal of inefficiencies without introducing semantic ambiguity among policies.

Chapter 3

Artificial Intelligence and Machine Learning

3.1 Overview of Artificial Intelligence

Artificial Intelligence (AI) comprises techniques and methods that enable machines and computer systems to carry out tasks that have traditionally required human cognitive abilities, such as reasoning, learning, perception, and decision making. The aim of AI is not merely to automate actions, but to build systems that can adapt, improve their performance, and make autonomous decisions on the basis of available data.

3.1.1 Artificial Intelligence vs. Machine Learning vs. Deep Learning

It is important to distinguish among several concepts that are often used interchangeably, but in fact stand in a hierarchical relationship:

- **Artificial Intelligence (AI)**: the broad field that encompasses all techniques that enable a computer to perform "intelligent" tasks.
- **Machine Learning (ML)**: a branch of AI focused on algorithms that fit models to data to make predictions or decisions and become more accurate over time.
- **Deep Learning (DL)**: a branch of ML that uses deep neural networks to automatically extract complex features and representations.

This relationship can be viewed as nested sets:

$$\text{AI} \supset \text{ML} \supset \text{DL}$$

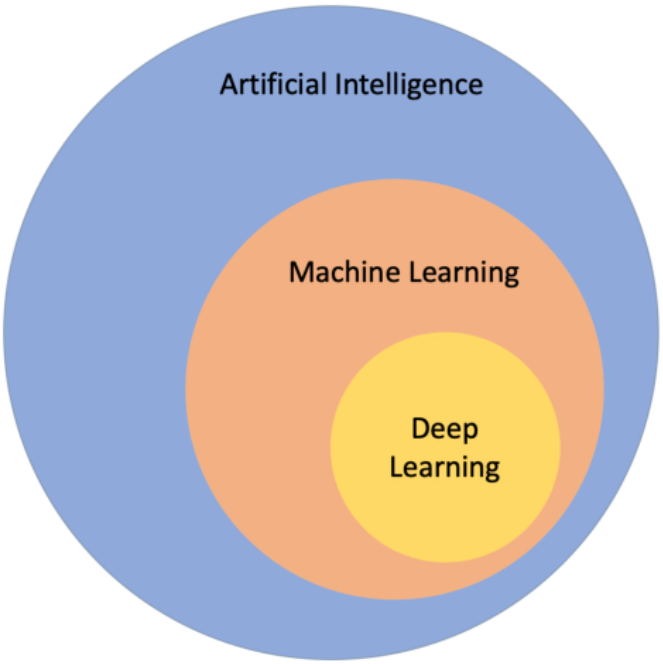


Figure 3.1. AI vs. ML vs. DL

3.1.2 The Role of Artificial Intelligence in Cybersecurity

In recent years, **Artificial Intelligence** (AI) has taken on an increasingly prominent role in cybersecurity, providing advanced tools for the detection, prevention, and response to cyber attacks. The threat landscape is rapidly evolving and more sophisticated, which has made it necessary to develop solutions that can analyse large volumes of data in real time and uncover complex patterns that are difficult to detect with traditional methods [15] [16].

The main applications of AI in this domain are several. First, it supports anomaly detection, spotting unusual patterns in network traffic or resource usage that can signal malicious activity. Then, it enables predictive analytics, where models learn from historical traces to anticipate likely threats by uncovering recurring patterns and correlations among events. Moreover, it drives the automation of incident response, shortening reaction times by allowing systems to apply countermeasures without human intervention when predefined conditions are met. It contributes also to the optimisation of defence systems, refining firewall rule sets, IDS and IPS configurations, and other security controls through learning algorithms that improve decisions as more evidence accumulates.

The integration of AI strengthens detection and mitigation capabilities, reducing false positives and accelerating threat analysis. Nevertheless, despite these clear benefits, there are significant limitations and challenges that constrain adoption on a large scale.

A first concern is limited transparency: models based on deep neural networks often behave as opaque systems, which complicates the explanation of outcomes and weakens the confidence of security analysts. A second issue is susceptibility

to adversarial inputs, since carefully crafted examples can induce incorrect behaviour. Moreover, some attacks intentionally imitate legitimate traffic, which makes it harder to separate benign activity from malicious activity. Fourth, many deployments still experience a substantial rate of false positives, which generates superfluous alerts, reduces operational efficiency, and undermines trust in automated defences. Finally, the reliance on large volumes of sensitive data raises ethical and privacy questions regarding governance, data protection, and regulatory compliance.

A further concern is the dual use character of artificial intelligence: the same techniques that support defence can also be repurposed for offensive objectives, enabling adversaries to create more sophisticated malware and design evasion strategies that slip past detection.

In conclusion artificial intelligence has become a strategic pillar of modern cybersecurity, strengthening both prevention and response. Its real impact, however, depends on responsible adoption: automation should be balanced with human control, and model transparency should be aligned with security objectives, so that the resulting solutions are robust, ethical, and sustainable over time.

3.2 Machine Learning Models

Machine Learning rests on an algorithm's ability to improve its performance through experience, that is, by analyzing and learning from historical data. Machine Learning models are primarily grouped into two categories: **Supervised Learning** and **Unsupervised Learning**.

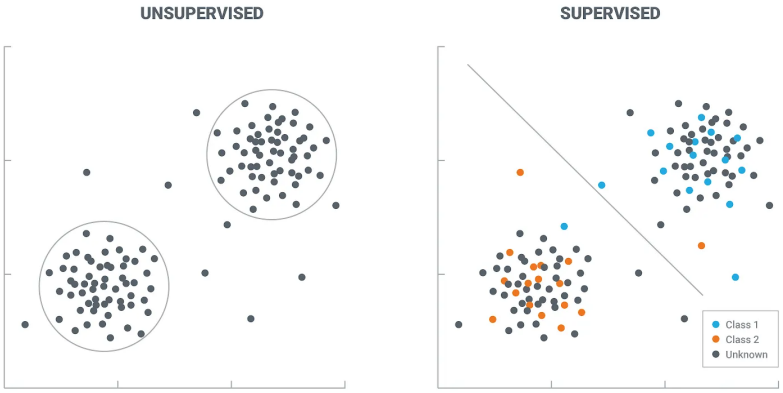


Figure 3.2. Supervised Learning vs. Unsupervised Learning

3.2.1 Supervised Learning

Supervised learning is a machine learning approach in which an algorithm learns to make predictions or classifications from a dataset of **labelled** examples. Each

training instance is paired with the correct answer, called the *label*.

For example, to build a system that recognises animals in images, the model is trained on images already labelled with the animal's name, such as "cat" or "dog". The algorithm analyses the features of these images and learns the relationship between the content and the assigned label.

Once the mapping between inputs and targets has been learned, the model can generalise to new cases it has not seen before. Training adjusts the parameters to minimise a loss function that measures the gap between predictions and the true labels in the data set; with labelled examples as feedback, accuracy typically improves over successive iterations.

Supervised learning spans two main task families: **classification**, which assigns an input to a discrete category (for example, deciding whether an email is spam), and **regression**, which estimates a continuous value (for instance, the price of a house). In cybersecurity, these techniques support spam filtering, malware identification, and the forecasting of network traffic patterns.

For firewall policy reordering, the problem can be framed in two ways:

1. **Multi-label classification**: each possible ordering of the rules is treated as a class. The model learns to map a given traffic profile to the best ordering. The main drawback is that the number of possible orderings grows factorially with the number of rules ($n!$), which makes the approach poorly scalable.

2. **Learning-to-Rank**: policy reordering is cast as a ranking problem in which the model assigns a score to each rule based on the likelihood that it should be placed near the top. This approach scales well and adapts to dynamic changes in the number of rules.

Learning-to-Rank Approaches

There are three main approaches:

- **Pointwise learning**: treats ranking as an independent classification or regression task for each item. It is simple but ignores relationships among items.

- **Pairwise learning**: produces a ranking by comparing pairs of items to decide which is more relevant. It captures pairwise relations but not the structure of the full list.

- **Listwise learning**: considers the entire list of items and directly optimises the quality of the final ranking.

In our setting, the *listwise* approach is preferable because it optimizes the global order and yields higher accuracy than the other two.

3.2.2 Unsupervised Learning

Unsupervised learning refers to methods that learn directly from data without labels or known answers, with the aim of discovering latent regularities, structures, or relationships. In practice, these techniques are often used to **cluster** observations into coherent groups, such as users that exhibit similar behaviour, and to perform **dimensionality reduction**, where the number of variables is reduced while retaining the most informative structure of the data.

In **cybersecurity**, unsupervised learning is used to detect *botnets* and to identify anomalies in network traffic. For firewall policy reordering, it is not suitable because it lacks an explicit optimization objective: it can reveal patterns, but it does not directly improve firewall performance.

3.3 Reinforcement Learning

Reinforcement learning (RL) is a machine learning paradigm in which an agent learns to make decisions by interacting with an environment, with the aim of maximising cumulative reward over time. Unlike *supervised learning*, where the algorithm learns from labelled data provided by an external supervisor, in reinforcement learning the agent does not receive explicit guidance about the correct action in each situation. Learning instead proceeds through trial and error: the agent tries different actions and receives a *reward* or a *penalty* according to the consequences of its choices.

A distinctive feature of RL is the handling of delayed reward: actions influence not only the immediate payoff but also the future state of the environment and, as a result, subsequent rewards. This creates the need to balance *exploration* (trying new actions to discover better strategies) and *exploitation* (using known actions that yield high rewards). The exploration versus exploitation tradeoff is a characteristic challenge of reinforcement learning and is absent from the "pure" forms of supervised and unsupervised learning.

From a theoretical standpoint, the RL problem can be formalized as the optimal control of a *Markov Decision Process* (MDP), possibly under partial observability. An agent perceives the state of the environment, selects actions that influence its evolution, and pursues an objective defined in terms of reward. In this sense, reinforcement learning is not only a sequential decision problem but also a research area that studies both the underlying models and the practical algorithms used to solve them, making it a paradigm that is distinct from, yet complementary to, supervised and unsupervised learning [17].

In **cybersecurity**, RL is used for automated cyber defence and for modelling adversarial attack and defence strategies.

3.3.1 Fundamental Elements of Reinforcement Learning

In *reinforcement learning*, learning does not rely on labelled examples but on a continuous cycle of interaction between an agent and an environment, where the

agent learns through trial and error. The main components of this paradigm are [18]:

- **Agent**: the decision making entity that acts on the basis of information received from the environment. It can be a robot that learns to navigate, software that plays chess, or an autonomous driving system.

- **Environment**: the context with which the agent interacts. It provides the agent with the current **state** and reacts to the agent's actions by changing that state.

- *Action*: the choice made by the agent in a given state. Actions determine how the environment evolves and influence future rewards.

- *Reward signal*: a numerical signal issued by the environment after each action, indicating the immediate benefit or cost of that choice. The reward defines what is desirable in the short term and is the primary criterion for updating the agent's policy.

- *Policy*: the strategy that maps states to actions. It can be deterministic (the same action for the same state) or stochastic (actions chosen with specified probabilities). The policy is the core of the agent's behaviour, since it entirely determines how the agent acts.

- *Value function*: quantifies how good a state (or a state paired with a specific action) is in terms of expected future reward. Unlike the reward, which reflects only the immediate outcome, the value function estimates the return over the long term under a given policy. It is a core component of most RL methods, guiding decisions towards actions that maximise cumulative gain.

- *Model of the environment*: an internal representation used by the agent to predict the environment's evolution, that is, the next state and reward given a state–action pair. Methods that use a model are called *model based* and can plan by simulating future scenarios, whereas *model free* methods rely solely on trial and error.

- **RL decision loop**: learning proceeds in a recurrent cycle

$$\text{state}_t \;\rightarrow\; \text{action}_t \;\rightarrow\; \text{reward}_{t+1},\ \text{state}_{t+1}$$

  which generates a sequence (s, a, r, s'). Through this sequence, the agent learns to maximise the *expected return*, namely the discounted sum of future rewards.

- **Episodic and continuing tasks**: RL problems may be episodic (with an initial and a terminal state) or continuing (without a natural end).

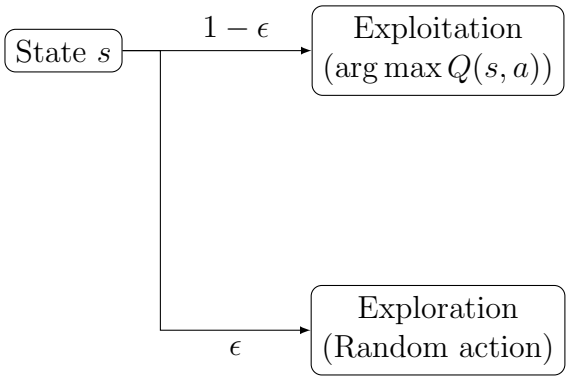
3.3.2 Exploration vs Exploitation

A central dilemma in reinforcement learning is the balance between **exploration** and **exploitation**.

- **Exploration**: the agent tries new actions, even if they have not historically yielded the highest rewards. The goal is to discover potentially better strategies or more profitable states.
- **Exploitation**: the agent chooses actions that, given current knowledge, promise the highest reward. This maximises short term gain but risks overlooking superior alternatives.

An agent that explores too much may never fully leverage what it has already learned; conversely, an agent that exploits too much may become stuck in a suboptimal strategy. A common way to balance this trade off is the ϵ**-greedy** strategy, in which:

$$a = \begin{cases} \text{random action} & \text{with probability } \epsilon, \\ \arg\max_a Q(s,a) & \text{with probability } 1-\epsilon. \end{cases}$$



3.3.3 Value-Based vs Policy-Based

Reinforcement learning algorithms are commonly grouped into two families:

Value-Based

The learner estimates a **value function** $V(s)$ or an action value function $Q(s,a)$, which approximates the expected cumulative reward from a state or a state–action pair. The optimal policy π^* is obtained by choosing the action with the highest value:

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

This class of methods is simple to implement and converges well in discrete spaces, but it is poorly suited to continuous action spaces without appropriate function approximation.

Policy-Based

The learner optimizes a parameterized policy $\pi_\theta(a \mid s)$ directly, using methods such as the **policy gradient**:

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

These methods naturally handle continuous action spaces and can represent stochastic policies, but they typically exhibit slower convergence and higher variance.

3.3.4 Monte Carlo vs Temporal Difference Learning

An additional important distinction concerns how value estimates are updated and, in particular, how bias and variance affect those estimates.

Monte Carlo (MC)

The Monte Carlo method updates values at the end of an episode using the observed cumulative return:

$$V(s) \leftarrow V(s) + \alpha\,[G_t - V(s)]\,,$$

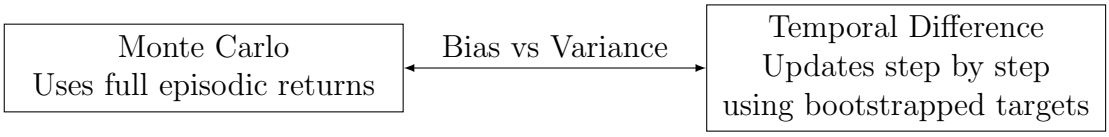
where $G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$. The target G_t is an **unbiased** estimate of $v_\pi(s)$, since by definition $v_\pi(s) = \mathbb{E}[G_t]$. However, because G_t depends on the entire sequence of rewards up to the end of the episode, it can vary widely across trajectories. This leads to **high variance**, which makes learning less stable.

Temporal Difference (TD)

The TD method updates estimates after each step, combining ideas from dynamic programming and Monte Carlo:

$$V(s) \leftarrow V(s) + \alpha\,[R_{t+1} + \gamma V(s_{t+1}) - V(s)]\,.$$

The TD target uses R_{t+1}, which is sampled from the environment, and $V(s_{t+1})$, which is only an **estimate** rather than the true value $v_\pi(s_{t+1})$. This introduces **bias**, because the quality of the update depends on the current accuracy of the value function. On the other hand, the variance is much lower than in MC, since the update considers only the immediate reward and the next state, which reduces uncertainty.



In summary, Monte Carlo methods provide **unbiased but high variance** estimates, whereas TD methods yield **lower variance but biased** estimates. As the value function improves, the bias in TD updates tends to diminish, balancing the two effects and making TD particularly effective for incremental learning.

3.3.5 Agent–Environment Loop

The operation of reinforcement learning can be summarised by the loop in Figure 3.3: the agent observes the state of the environment, selects an action, receives a reward, and then observes the next state. This process repeats until the episode terminates, or indefinitely in continuing environments.

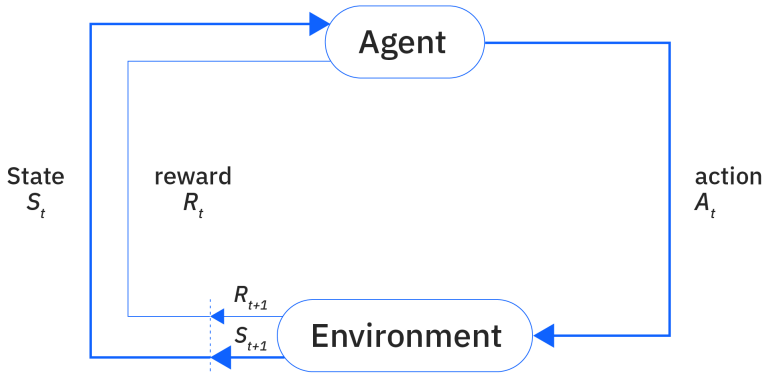


Figure 3.3. Schematic of the agent–environment cycle in reinforcement learning

3.4 Main Reinforcement Learning Algorithms

In this section we examine three foundational algorithms: the **k-armed bandit** problem, **SARSA**, and **Q-Learning**. The latter two belong to the family of value-based methods, whereas the k-armed bandit offers a simplified model that introduces core ideas such as the exploration–exploitation trade off.

3.4.1 k-armed Bandit

The **k-armed bandit** is a classical sequential learning problem that compactly captures the **exploration–exploitation trade off**. The name comes from the analogy with a slot machine (*one-armed bandit*) that has k levers, each associated with a different reward distribution.

Problem Definition

- **Action**: at time step t, the agent selects an action $A_t \in \{1, \ldots, k\}$.
- **Reward**: the agent receives a numerical payoff R_t drawn from a stationary distribution that depends on the chosen action.
- **Objective**: maximise the expected sum of rewards over a horizon of T steps.

The **true value** of an action a is

$$q^*(a) \doteq \mathbb{E}[R_t \mid A_t = a],$$

while the agent maintains an **estimate** $Q_t(a)$ that is updated from experience.

Incremental Learning Update

A simple update rule for the action value estimate is:

$$Q_{t+1}(A_t) \leftarrow Q_t(A_t) + \alpha_t \big[R_t - Q_t(A_t)\big],$$

where α_t is the learning rate (for example, $\alpha_t = \frac{1}{N_t(A_t)}$, with $N_t(a)$ the number of times action a has been selected up to time t).

Basic Load Balancing Strategies in Reinforcement Learning

- **ϵ-greedy**: with probability ϵ select a random action (exploration), otherwise choose the greedy action (exploitation).
- **Pure greedy**: always exploit; this can be suboptimal if the initial estimates are inaccurate.
- **Initial exploration**: perform an initial phase of random choices, then switch to pure exploitation.

Despite its simplicity, the k-armed bandit highlights a central challenge of reinforcement learning: deciding **how much to explore and when** in order to improve current knowledge without sacrificing too much immediate reward.

3.4.2 SARSA and Expected SARSA

SARSA (*State–Action–Reward–State–Action*) is an **on policy** control algorithm based on **Temporal Difference** learning. Unlike Q learning, SARSA updates the estimate $Q(s, a)$ using the action actually taken in the next state, thereby reflecting the behaviour of the policy followed during learning.

Learning Update

The standard SARSA update is:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma\, Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)\right],$$

where:

- a_{t+1} is the action actually executed in state s_{t+1} under the current policy (e.g., ϵ greedy).

- α is the learning rate, $0 < \alpha \leq 1$.
- r_{t+1} is the immediate reward obtained when moving from s_t to s_{t+1} after taking a_t.
- γ is the discount factor that sets the importance of future rewards, $0 \leq \gamma < 1$.
- $Q(s_{t+1}, a_{t+1})$ is the estimated action value in the next state s_{t+1} for the action a_{t+1} chosen by the current policy (e.g., ϵ greedy).

Main Features

- **On policy**: learns action values for the policy being followed, including the effects of exploration.
- More conservative than Q learning in risky environments, since the bootstrapped target does not always assume the maximum over actions.
- May be less performant in the long run if the policy explores heavily, but it avoids risky behaviour.

Expected SARSA

Expected SARSA replaces $Q(s_{t+1}, a_{t+1})$ with the expectation under the action distribution in state s_{t+1}:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \sum_a \pi(a \mid s_{t+1})\, Q(s_{t+1}, a) - Q(s_t, a_t) \right].$$

- Reduces the variance of the update by removing the randomness due to a single sampled action a_{t+1}.
- Often outperforms SARSA, especially with larger learning rates α.
- Can be used in both on policy and off policy settings, providing a bridge between SARSA and Q learning.

3.4.3 Q-Learning

Q-learning is one of the most influential algorithms in reinforcement learning. It is an **off policy** control method based on **Temporal Difference** (TD) learning. Introduced by Watkins in 1989, it marked a turning point because it allows the agent to estimate the optimal action value function $q^*(s, a)$ directly, independently of the behaviour policy followed during learning.

Q-table

The main data structure in Q-learning is the **Q-table**, a matrix whose rows correspond to states and whose columns correspond to available actions. Each entry $Q(s, a)$ is an estimate of the expected return obtained by taking action a in state s and then following the optimal policy:

$$Q(s, a) = \text{expected future return.}$$

Learning Update

After each interaction, the agent updates the Q-table using the received reward and the observed next state:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right],$$

where:

- $Q(s, a)$: estimate of the value of the state–action pair.
- α: learning rate, $0 < \alpha \leq 1$.
- r_{t+1}: immediate reward obtained after taking a_t in state s_t.
- γ: discount factor, $0 \leq \gamma < 1$.
- $\max_a Q(s_{t+1}, a)$: best estimated value in the next state, maximised over all actions.

Main Features

- **Off policy**: learns the value of the optimal policy even while following an exploratory policy (for example, ϵ-greedy) to gather data.
- **Convergence**: proven to converge to q^* with probability one [19], provided that all state action pairs are visited infinitely often and the learning rates α satisfy the standard stochastic approximation conditions.
- **Simplicity**: it does not require an explicit model of the environment.

3.5 From Background to the Present Work

3.5.1 Reinforcement Learning vs. Listwise Learning

Model selection must weigh the advantages and limitations of the two approaches. Listwise learning trains on historical data to predict the optimal ordering of rules. The model learns to rank the entire rule list so as to improve firewall efficiency. It is often simpler to implement than reinforcement learning and produces results

grounded in observed data. However, its effectiveness depends on the quality and representativeness of the training set. In addition, it may not adapt quickly to abrupt changes in network traffic, since it relies on patterns learned from past data, which makes it less flexible and more static.
Reinforcement learning uses an agent that explores alternative orderings of the rules and evaluates the effectiveness of each configuration. This process enables dynamic adaptation to changes in network traffic, thereby improving firewall performance over time. However, reinforcement learning can be complex to implement and requires a training period during which the agent may not operate optimally.
In conclusion, given the initial objective of achieving dynamic reordering, reinforcement learning is the more appropriate choice.

Chapter 4

Thesis Objectives

Firewalls constitute a primary line of defence for protecting networks, as they regulate communication flows by determining which packets may cross the perimeter and which must be blocked. This function is realised through an ordered set of filtering rules that are evaluated sequentially until a match is found.

Such mechanism can become very inefficient as the number of rules grows. In those cases, the rise in complexity slows the packet checking and increases the probability of poorly performing configurations. A firewall that does not manage its policy correctly can introduce significant delays in traffic handling and, in the worst cases, leave systems exposed to vulnerabilities that attackers may exploit.

With a rule configuration that is non optimal it's possible to have a lot of *misses*, namely packets that are not handled as intended because of inadequate ordering. In complex networks with high traffic volumes, these issues undermine both overall service reliability and the level of security provided by the protection system.
This scenario highlights the need for a more intelligent and adaptive approach to firewall policy management.

Goal of the Solution

In this thesis we want to develop an **intelligent firewall** capable of dynamically optimizing the ordering of filtering rules, overcoming the limitations of static approaches that typically rely on simple sorting techniques. The proposal is, in particular, to employ a **Reinforcement Learning** algorithm, which enables an agent to learn from observed network traffic and to assign a distinct value to each rule. In this way, the most relevant rules are brought to the top of the list, reducing processing time and potential *misses*.

To realise this vision, the work designs a **learning agent** that interacts with the firewall: it analyses incoming packets, updates rule evaluations on the fly, and periodically supplies the traditional firewall with an optimised ordering of the policy. Once the traffic pattern has been learned, the agent operates alongside the firewall, ensuring continuous and adaptive reordering.
In summary, the system aims to:

- Reduce latency introduced by the filtering process;

- Minimise *misses* due to ineffective orderings;
- Improve overall network reliability and security;
- Introduce a dynamic and adaptive approach that reacts to traffic changes in real time.

Chapter 5

Approach and Methodology

5.1 Smart Firewall

As outlined in the previous chapters, the goal is to design an intelligent agent that behaves as an **adaptive firewall**, able to select dynamically the most appropriate rule to apply to each incoming packet. The agent learns through **Reinforcement Learning**, in particular via the **Q-Learning** algorithm, and progressively improves its decisions on the basis of accumulated experience. The idea is for the agent to discover, through interaction, which rules are most effective given the observed traffic.

In the Q-table, each state represents a packet to be classified, while actions correspond to selecting one of the rules available in the firewall. Initially all rules have value Q = 0.

When a packet arrives, the agent selects a rule with an ϵ-greedy policy. If the selected rule matches the packet, it receives a **positive reward** and the value of that rule is updated; otherwise, it incurs a **penalty** and proceeds to the next candidate, choosing among the remaining rules the one with the highest current Q value while excluding those already attempted.

The Q-table is updated using the standard Q-Learning update rule, in addition, a **time decay** mechanism lowers the value of rules that are not used for a certain number of consecutive packets, thereby modelling the gradual obsolescence of rules over time.

This mechanism allows the agent to adjust its choices as it learns, improving filtering effectiveness and keeping the decision policy in step with changing traffic patterns.

Example

We consider three initial rules with $Q = 0$. Packets arrive sequentially and the agent updates the Q-table according to the received rewards. We use reward $r = 1$ and penalty $r = -1$, with $\alpha = 0.5$.

Initial setup

- Firewall rules:
  - `Rule_A`
  - `Rule_B`
  - `Rule_C`
- Initial Q-table:

Rule	Q-value	Last used
Rule_A	0	–
Rule_B	0	–
Rule_C	0	–

Table 5.1. Initialization of Q values and rule usage state

- Parameters:
  - $\alpha = 0.5$
  - $\gamma = 0$
  - $\epsilon = 0.2$
  - Reward: $+1$ if the selected rule is correct, -1 otherwise
  - Decay: after every 5 packets of inactivity, $Q \leftarrow Q \times 0.9$

Evolution of the Q-table: Packets 1 to 5

Packet 1: associated with `Rule_B`.
`Rule_A` is selected (greedy) but is incorrect:

$$Q_A = 0 + 0.5 \cdot (-1) = -0.5.$$

Then `Rule_B` is selected and is correct:

$$Q_B = 0 + 0.5 \cdot 1 = 0.5.$$

Rule	Q-value
Rule_A	-0.5
Rule_B	0.5
Rule_C	0

Table 5.2. Q-table after the first packet

Packet 2: again associated with `Rule_B`.
`Rule_B` is selected (exploitation) and is correct:

$$Q_B = 0.5 + 0.5 \cdot (1 - 0.5) = 0.75.$$

Rule	Q-value
Rule_A	-0.5
Rule_B	0.75
Rule_C	0

Table 5.3. Q-table after the second packet

Packet 3: associated with `Rule_C`.
`Rule_B` is selected (exploitation) but is incorrect:

$$Q_B = 0.75 + 0.5 \cdot (-1 - 0.75) = -0.125.$$

Then `Rule_C` is selected and is correct:

$$Q_C = 0 + 0.5 \cdot 1 = 0.5.$$

Rule	Q-value
Rule_A	-0.5
Rule_B	-0.125
Rule_C	0.5

Table 5.4. Q-table after the third packet

Packet 4: again associated with `Rule_C`.
`Rule_C` is selected (exploitation) and is correct:

$$Q_C = 0.5 + 0.5 \cdot (1 - 0.5) = 0.75.$$

Rule	Q-value
Rule_A	-0.5
Rule_B	-0.125
Rule_C	0.75

Table 5.5. Q-table after the fourth packet

Packet 5: associated with `Rule_A`.
`Rule_C` is selected (exploitation) but is incorrect:

$$Q_C = 0.75 + 0.5 \cdot (-1 - 0.75) = -0.125.$$

Then `Rule_A` is selected and is correct:

$$Q_A = -0.5 + 0.5 \cdot \big(1 - (-0.5)\big) = 0.25.$$

Rule	Q-value
Rule_A	0.25
Rule_B	-0.125
Rule_C	-0.125

Table 5.6. Q-table after the fifth packet

Packets 6 to 10 (summary)

Over packets 6 to 10, the agent observes the following ground truth sequence: `B`, `B`, `C`, `C`, `C`. The choices yield the following updates (we report only the final values):

$$Q_A = 0.25, \quad Q_B = -0.140625, \quad Q_C = 0.859375.$$

The *last used* packets are: `Rule_A` at packet 5, `Rule_B` at packet 8, and `Rule_C` at packet 10.

Decay after Packet 10

From packet 11 onward, if a rule has not been used in the last 5 packets, its Q value is reduced (*times* 0.9). Since `Rule_A` was not used between packets 6 to 10, it decays; `Rule_B` and `Rule_C` do not.

Rule	Q before	Last used	Action
Rule_A	0.25	5	Decays $\rightarrow 0.25 \times 0.9 = 0.225$
Rule_B	-0.140625	8	No decay
Rule_C	0.859375	10	No decay

Table 5.7. Q-value updates with decay applied (threshold: 5 packets of inactivity)

Configurations

The development of an intelligent firewall based on Reinforcement Learning proceeded through three main stages: an initial *stateless* model, the gradual introduction of *state* to characterize packets, and, finally, the design of a configuration with a digital twin.

5.1.1 Baseline Firewall (Stateless)

The first implementation adopted a deliberately simple structure: an agent interacting with an environment without state. At each packet arrival, the agent applies

an ϵ-greedy policy to select one of the available actions. If the chosen action is incorrect, the agent attempts another action, distinct from those already tried for that specific packet, penalising incorrect choices and rewarding correct ones. This approach produced a working prototype, but it suffers from significant limitations due to the absence of a packet state representation.

5.1.2 Firewall with State Representation

To overcome the limitations of the previous model, we introduced the notion of *state* into the decision process. Whereas the first firewall handled all packets in the same way, here each packet is associated with a state based on its characteristics, so that optimal actions can vary with the state under consideration. This increases complexity: the Q-table is no longer a single row with n columns (where n is the number of rules), but there are m rows, corresponding to the states, per n columns. To limit table growth, we consider at most 27 states, focusing in particular on configurations with 3, 9, and 27 states.

Firewall with 3 States

In this configuration, the state is determined solely by the packet's protocol:

- TCP
- UDP
- other protocols

For example:

`192.168.1.1 8.8.8.8 102 8080 TCP` → `state TCP`
`192.168.1.1 132.4.4.1 102 65000 UDP` → `state UDP`

Firewall with 9 States

Here we extend the previous classification by combining the three protocols (TCP, UDP, other) with a three-way split of the *source port* [20]:

- **Well known ports** (0–1023): reserved by the operating system or server applications (e.g., FTP 21, SSH 22, SMTP 25, HTTP 80).
- **Registered ports** (1024–49151): used by user applications for communications that are specific.
- **Dynamic ports** (49152–65535): assigned to client applications dynamically.

Combining the two classifications yields $3 \times 3 = 9$ distinct states.
For example:

`192.168.1.1 8.8.8.8 102 8080 TCP` → `state TCP-W (source well known)`
`192.168.1.1 132.4.4.1 55000 65000 UDP` → `state UDP-D (source dynamic)`

Firewall with 27 States

The final configuration further extends the model by also splitting the *destination port* into three ranges. The state space thus becomes 3 (protocols) × 3 (source ports) × 3 (destination ports), for a total of 27 states.
For example:

`192.168.1.1 8.8.8.8 102 8080 TCP → state TCP-W-R (source well known, destination registered)`
`192.168.1.1 132.4.4.1 55000 80 UDP → state UDP-D-W (source dynamic, destination well known)`

5.1.3 Digital Twin configuration

In the final configuration we adopt a hybrid approach that runs two distinct firewalls in parallel on the same packet stream. The first is the *smart firewall*, described in the previous sections, which can be implemented either in a *stateless* mode or with explicit states, according to the administrator's needs. The second is a traditional firewall that applies the *first matching rule* resolution strategy.

The overall operation works in this way: each incoming packet is processed in parallel by both firewalls. The *smart firewall* does not directly decide whether to accept or drop the packet; instead, it learns from the traffic and updates the rule list dynamically. Periodically, or after a specified volume of traffic, the *smart firewall* produces a newly ordered rule set, optimized on the basis of the observed patterns, and transfers it to the traditional firewall. The latter is responsible for the final decision (`ACCEPT`/`DENY`) on packets, while benefiting from a policy that is continuously updated and adapted to current traffic.

In this way, the traditional firewall preserves its established, reliable structure, while the *smart firewall* acts as an optimization engine that reduces *misses* and improves overall performance, particularly in terms of latency.

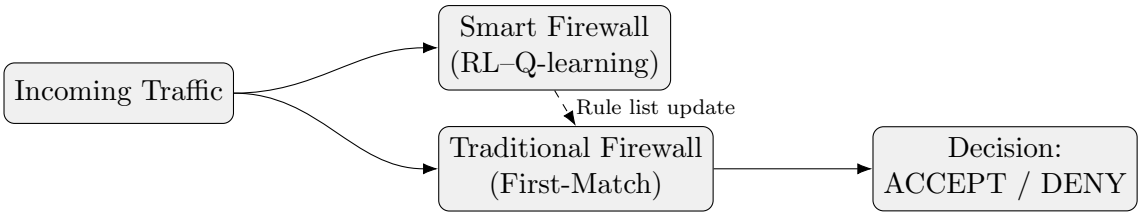


Figure 5.1. Digital twin architecture

5.2 Rule Generator

Before starting the implementation and validation phases of our intelligent firewall, we precisely specified the required **inputs** to simulate system behaviour. In particular, we considered two fundamental elements: the network traffic, represented

in the five tuple format `(srcIP, dstIP, srcPort, dstPort, protocol)`, and a rule set that is coherent with, and comprehensive of, that traffic.

As a first step we used **ClassBench**, a widely used tool for generating synthetic IPv4 rule sets in the five tuple format. ClassBench can analyse an existing rule set to extract a *SEED*, or generate a synthetic rule set from a given parameter file. In our case, we used it to produce several rule sets with the following command:

```
./vendor/db_generator/db_generator -bc vendor/parameter_files/fw1_seed
   100 2 -0.5 0.1 output.rules
```

The parameters have the following meaning:

- `-b`: enables scaling of IP prefixes to obtain a more realistic distribution.
- `-c vendor/parameter_files/fw1_seed`: selects the parameter file that guides the type of rules generated (in this case, firewall rules).
- `100`: number of rules to generate.
- `2`: *smoothness* parameter that controls structural variety in the rules.
- `-0.5`: *address scope* parameter that sets the specificity of IP prefixes (negative values yield more specific prefixes).
- `0.1`: *port scope* parameter, analogous to the above but for ports (positive values yield wider ranges).
- `output.rules`: output file that will contain the generated rule set.

After obtaining the synthetic rules, we defined in Python a sampling scheme for packet creation so as to generate artificial traffic consistent with the produced rules.

To increase the robustness and validity of testing, we also selected a real world dataset. Specifically, we used the **SCADA/ICS Network Captures** dataset, which contains about 239,000 packets with TCP, UDP, and ICMP as the main protocols (Figure 5.2).

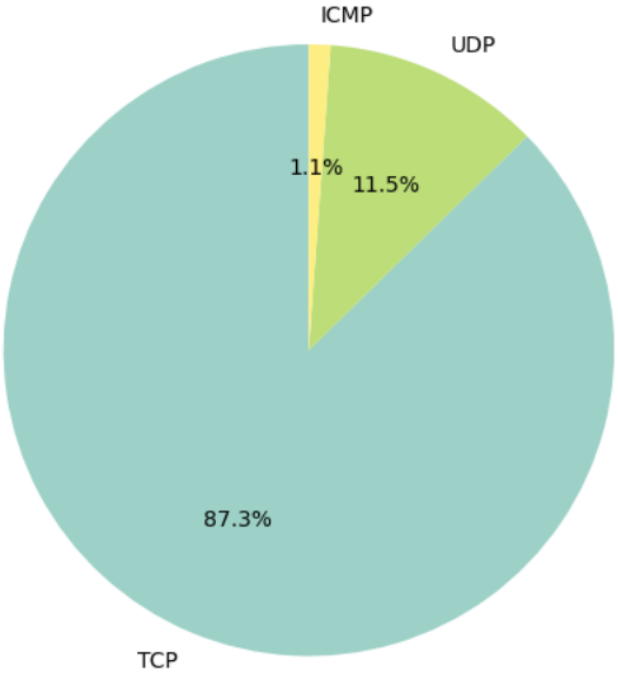


Figure 5.2. Distribution of packets per protocol

The use of a real world dataset made it necessary to develop a **dedicated rule generator** able to take the observed packets as input and produce a rule set that covers every packet at least once. To build this generator, we adopted an **unsupervised learning** approach aimed at discovering recurrent patterns in the traffic and translating them into filtering rules. In particular, we focused on three clustering algorithms:

- **k-means**, which partitions packets into a predefined number of clusters.
- **DBSCAN**, which identifies clusters of arbitrary density and separates noise.
- **Hierarchical clustering**, which constructs a hierarchy of clusters and enables multi level analysis of the rules.

5.2.1 K-Means Clustering

K-Means is one of the most widely used clustering algorithms in data science and machine learning. It is a *partitional clustering* method that divides an unlabelled dataset into K distinct, non-overlapping groups. Each group is represented by a *centroid*, the mean of the points assigned to that cluster. The algorithm seeks to minimize intra-cluster distance, so that items within the same group are as similar as possible, while being as dissimilar as possible from items in other groups.
The method was first introduced by Hugo Steinhaus in 1956 and later formalised by MacQueen in 1967, becoming a standard tool in statistical analysis and, more recently, in machine learning.

Operation

The algorithm proceeds through an iterative sequence of steps [21]:

1. Specify the desired number of clusters K.
2. Select K initial points at random as provisional centroids.
3. Assign each data point to the cluster whose centroid is closest (Figure 5.3), using a distance metric—typically Euclidean distance, though alternatives such as cosine similarity or correlation are possible.
4. Recompute each centroid as the mean of the points assigned to that cluster.
5. Repeat steps 3 and 4 until convergence, that is, until assignments no longer change substantially or a maximum number of iterations is reached.

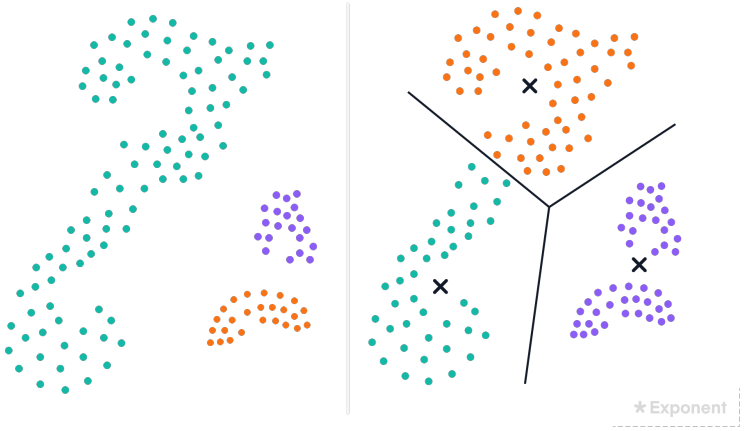


Figure 5.3. Examples of *K-Means*

The computational complexity is $O(n \cdot K \cdot I \cdot d)$, where n is the number of points, K the number of clusters, I the number of iterations, and d the data dimensionality. In practice, most of the convergence typically occurs in the early iterations.

Among its strengths, *K-Means* is simple to understand and implement; on data sets with high dimensional features it can be more efficient than hierarchical methods, and it produces partitions that are easy to interpret. However, it also has notable limitations: the number of clusters K must be fixed in advance, the outcome is sensitive to the initial centroids and to the order in which the data are processed; performance is best when clusters are roughly spherical and of similar size, and degrades on data with more complex structure; and its sensitivity to feature scaling makes data normalisation advisable before use.

5.2.2 DBSCAN

DBSCAN (*Density-Based Spatial Clustering of Applications with Noise*) is a density-based clustering algorithm designed to identify groups of points in high-density

regions separated by low-density areas. Unlike centroid-based approaches such as K-Means, DBSCAN does not require the number of clusters to be specified in advance and is particularly effective at discovering arbitrarily shaped structures while identifying noise and outliers.

Operation

The algorithm relies on two main parameters [22]:

- ε (eps): the neighbourhood radius within which a point considers its neighbours;
- MinPts: the minimum number of points required within the ε radius for a region to be deemed sufficiently dense to form a cluster.

Given these parameters, DBSCAN distinguishes three types of points (Figure 5.4):

- **Core points**: points with at least MinPts neighbours within radius ε. They lie in dense areas and form the core of clusters.
- **Border points**: points with fewer than MinPts neighbours that nevertheless fall within the ε neighbourhood of a core point. They belong to the cluster but lie on its boundary.
- **Noise points**: points that satisfy neither condition and therefore do not belong to any cluster. These are typically treated as outliers or anomalies.

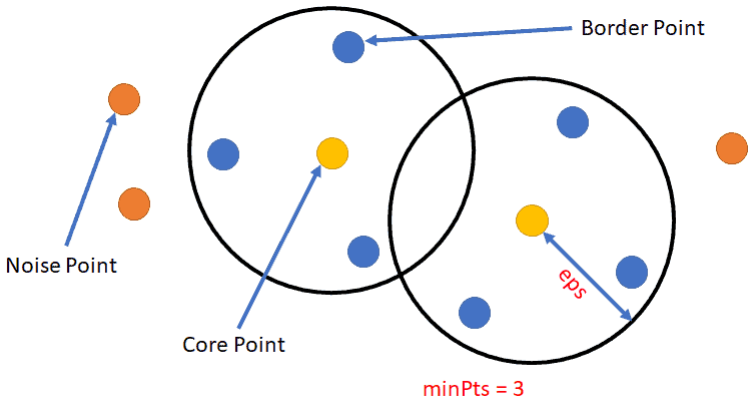


Figure 5.4. Examples of *DBSCAN*

The algorithm explores the dataset starting from a core point and progressively includes all *density reachable* points, that is, points reachable through a chain of sufficiently dense neighbours. In this way, DBSCAN can recover clusters of arbitrary shape rather than being limited to spherical structures.

This method has several strengths: it does not require choosing the number of clusters K in advance; it can discover clusters of arbitrary shape rather than only

circular or elliptical groups, and it naturally flags outliers by assigning them to a noise set. At the same time, its performance is highly dependent on the choice of ε and MinPts—poor settings may yield too many noise points or merge distinct groups into overly large clusters. It also struggles on data sets with widely varying densities, where some clusters are very dense and others sparse, and its efficiency can degrade in high dimensional spaces because meaningful distance measures are harder to define.

5.2.3 Hierarchical clustering

Hierarchical Clustering is a clustering technique that builds a hierarchy of clusters, typically represented as a tree (a dendrogram). Unlike partitional algorithms such as K-Means, it does not require the number of clusters to be specified in advance, since the hierarchical structure allows groupings to be inspected at different levels of granularity [23].
There are two main approaches (Figure 5.5):

- **Agglomerative** (bottom up): each data point starts as a single cluster; at each iteration, the most similar clusters are merged until a single cluster containing all points is formed.
- **Divisive** (top down): the entire dataset is initially treated as one cluster, which is then progressively split into subclusters down to elementary groups.

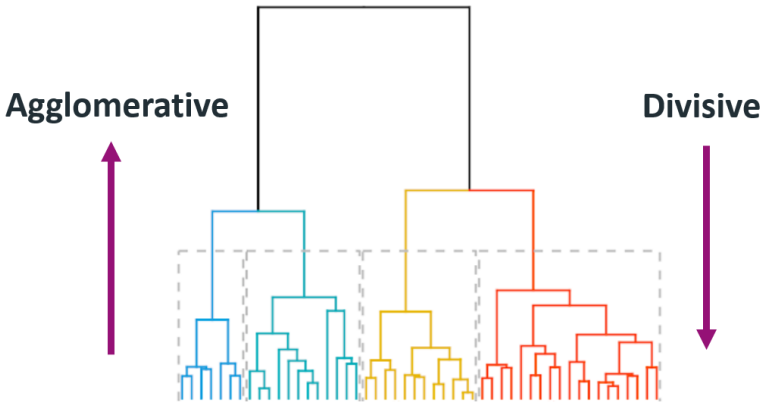


Figure 5.5. Examples of *Hierarchical Clustering*

Operation

The hierarchical process relies on a similarity measure (e.g., Euclidean distance, cosine, or correlation) and on a *linkage* strategy that defines how distances between clusters are computed [24]:

- **Single linkage**: minimum distance between two points belonging to different clusters.

- **Complete linkage**: maximum distance between points belonging to different clusters.
- **Ward's method**: merges clusters to minimize the increase in within-cluster variance.

The final output is a dendrogram that shows the sequence of merges (or splits) and allows the desired number of clusters to be chosen by cutting the tree at a selected height.

Hierarchical clustering is appealing in exploratory work because it does not require fixing the number of clusters in advance; the structure emerges from the data. It also yields a dendrogram that analysts can read to examine groupings at different levels of detail. On the downside, the method is sensitive to outliers, which can skew the hierarchy, and decisions taken early in the process, such as merges or splits, cannot be reversed, so early mistakes may propagate and lead to suboptimal partitions.

In light of the foregoing considerations, we decided to exclude **DBSCAN**. The goal of the rule generator is to ensure that every packet is assigned to at least one cluster and, consequently, to a corresponding rule. However, DBSCAN may label some packets as *noise points*, which leaves them outside the main clusters.

An initial strategy considered merging all *noise points* into a single cluster and deriving a generic rule from it. This approach would produce an overly broad rule with no meaningful common characteristics among the aggregated packets, thereby undermining the effectiveness and granularity of the clustering process.

For these reasons, the final choice was to implement two distinct generators: one based on **K-Means** and the other on **hierarchical clustering**, which can guarantee complete coverage of packets without introducing inefficiencies due to rules that are too general.

5.2.4 K-Means Rule Generator

This section describes the operation of the first rule generator, which is based on the K-Means clustering algorithm.

Preprocessing and feature engineering

Preprocessing was required to handle the heterogeneous nature of network traffic. In this phase, we performed *feature engineering* on both IP addresses and ports. For each protocol we estimate **IP diversity** (the number of unique `srcIP,dstIP` pairs and the cardinality of individual sources and destinations). Based on this measure, the generator automatically selects one of the following representations:

- **Original** (8 features): normalised `srcIP`/`dstIP` and low granularity hierarchical encodings (`/24`, `/16`, `/8`). Recommended in low diversity settings, it reduces the risk of overfitting.

- **Hybrid** (22 features): intermediate subnet granularity (`/32`–`/4`) with *decreasing weights*, augmented with relational features (same subnet, IP proximity). Suitable for medium diversity scenarios.

- **Hierarchical** (68 features): a multi level representation of the IP address that includes all subnets from `/32` down to `/1`. Each level is weighted hierarchically and enriched with relational features so as to capture complex patterns and relationships at multiple granularities. This configuration is preferred in high diversity contexts where maximum separability is required.

For TCP/UDP we add further port related features:

- **normalised values**: `srcPort/65535`, `dstPort/65535`;

- **one hot encoding** of IANA categories (*well known, registered, dynamic*) for source and destination;

- **relational features**: same category (binary) and *proximity* computed as $1 - \frac{|src-dst|}{65535}$.

All features are then standardised with `StandardScaler` ($\mu = 0$, $\sigma = 1$) before clustering. This prevents any variable from dominating the Euclidean distance due to scale differences, a problem to which K-Means is particularly sensitive.

Per-protocol clustering

Clustering is applied separately for each protocol through the following steps:

1. **Determining the number of clusters**: the target number of rules assigned to each protocol is proportional to its traffic share. For example, if the overall goal is to generate 100 clusters and TCP traffic accounts for 20% of the total, TCP is assigned 20 clusters. From this value, we subtract a quota reserved for generating rules with *partial overlap* and *full overlap*, which yields the effective number of clusters to create. The final value is also bounded by the number of unique samples available.

2. **Learning**: K-Means is applied with `n_init=10` and `random_state=42`. Using multiple initialisations reduces the risk of converging to suboptimal solutions due to poor centroid initialisation.

3. **Internal evaluation**: where applicable, partition quality is estimated with the *silhouette score*.

The proportional allocation avoids bias among protocols, while introducing controlled overlaps produces more realistic rule sets that are useful during validation.

Rule synthesis from clusters

Given the clusters, firewall rules are synthesized as follows:

1. **IP ranges**: compute the minimum and maximum of `srcIP_int` and `dstIP_int`. For the destination, compute the smallest **CIDR** network that covers both endpoints (`calculate_cidr_network`). Apply the same logic to the source.
2. **Port ranges**: for TCP/UDP, use the minimum and maximum of `srcPort` and `dstPort` (formatted as a range or a single value). For protocols without ports, set the range to 0–65535.
3. **Base rule**: generate a rule `(srcIP, dstIP, srcPort, dstPort, protocol)` and attach the number of covered packets as a statistical weight.

Controlled overlaps

For a predefined percentage of clusters, two distinct rules are generated:

- **Partial overlap**: one nontrivial ranged field (`srcIP`, `dstIP`, `srcPort`, or `dstPort`) is split into two subranges with a 60%/40% split. The two resulting rules share the remaining fields and partially overlap.
- **Full overlap**: create a "*small*" rule (50% subrange) and a "*full*" rule (entire range), so the former is fully contained in the latter.

This choice allows us to simulate *redundancy* and *shadowing* phenomena commonly found in real firewall configurations. In our experiments the parameters are set to 20% of rules with partial overlap and 10% with full overlap, though they can be disabled entirely to obtain anomaly-free rule sets.

Advantages of the approach

The approach brings several advantages. It is **adaptive**, with an automatic selection of features, informed by the diversity of the data, allows the system to adjust to new contexts while remaining robust. It **scales** well because clustering is carried out per protocol, which accommodates heterogeneous data sets from small collections to large enterprise networks. It also increases **realism**, in fact the rule generator introduces controlled overlaps so that the resulting policy reproduces dynamics commonly seen in practice. Finally, it improves **efficiency**, so running K-Means with multiple initializations lowers the risk of poor local optima and yields more stable partitions.

5.2.5 Hierarchical Rule Generator

This section analyses the second configuration of the rule generator, based on a **top down** hierarchical approach. Unlike a purely partitional method, the pipeline adopts **multi phase** hierarchical clustering organized as follows:

1. grouping of **IP pairs** (`srcIP, dstIP`);
2. optional subdivision by **port pairs** (`srcPort, dstPort`) within the IP clusters;
3. a **post adjustment** phase to align the number of clusters (and thus rules) with the desired target *exactly*;
4. synthesis of the **firewall rules** (CIDR plus port ranges), with the option to introduce *controlled overlaps* (partial and full).

As in the K-Means approach, the process is conducted **per protocol** (TCP, UDP, other), ensuring semantic and statistical consistency across the data.

Preprocessing and feature engineering

Preprocessing follows the same adaptive logic as in the first generator. For IP addresses we use hierarchical representations (for example, subnets `/24`, `/16`, ...), while for ports we consider numerical distance and the IANA semantics (*well known*, *registered*, *dynamic*). All features are standardised with `StandardScaler` so as to mitigate the effect of differing numeric scales on the Euclidean distances used during clustering.

Cluster creation process

For each protocol, the target number of rules is proportional to its share of packets. A percentage is reserved for generating rules with *controlled overlaps*, and the remainder determines the number of clusters that are actually "needed". The target is also bounded by the number of unique samples available, to avoid over partitioning.

Phase 1: Clustering of IP pairs
We extract unique (`srcIP, dstIP`) pairs and apply an **adaptive K-Means** on the selected IP features (Original/Hybrid/Hierarchical). The initial number of clusters is set to about 75% of the final target, limited by the number of available pairs. We explore a neighbourhood around the requested cluster count and select the configuration with the best *silhouette score*, adjusted with a penalty for deviations from the target ("*adjusted silhouette*").

Phase 2: Subdivision by port pairs (TCP/UDP)
For port based protocols, each IP cluster can be further split into **port pair** clusters by applying K-Means to the (`srcPort, dstPort`) features. The decision to split depends on a *split priority* measure, computed as a weighted combination of three main factors:

- **Spread**: measures the dispersion of port pairs using the average Euclidean distance among them.

  - *Low spread example:* `[(80,443), (81,444), (82,445)]`. The ports are consecutive, with a normalised average distance of ≈ 0.1. The cluster is homogeneous and does not require further splitting.
  - *High spread example:* `[(22,22), (443,443), (8080,8080), (65000,65000)]`. The pairs span very distant intervals, with spread ≈ 0.8. The cluster is heterogeneous and benefits from a split.

- **Range span**: captures the overall span of source and destination ports.
  - *Low range span example:* `[(80,443), (8080,8443)]`. The combined interval covers about 12% of the port space, indicating a relatively compact cluster (standard HTTP and variants).
  - *High range span example:* `[(22,22), (443,443), (3389,3389), (65000,65000)]`. The interval covers almost the entire port space (normalised value ≈ 1.0). The cluster is very diverse and should be prioritised for splitting.

- **Consecutiveness**: evaluates how consecutive the port pairs are.
  - *Low consecutiveness example:* `[(22,22), (443,443), (8080,8080), (65000,65000)]`. No pair is consecutive, so consecutiveness is 0. The cluster is highly heterogeneous and a good candidate for further division.
  - *High consecutiveness example:* `[(80,80), (81,81), (82,82), (83,83), (90,90)]`. Three out of four adjacent steps are consecutive, with consecutiveness = 0.75. The cluster is homogeneous and may not need a split.

The final priority is computed as:

$$P = 0.3{\cdot}\text{Spread}+0.3{\cdot}\text{RangeSpan}+0.2{\cdot}(1-\text{Consecutiveness})+0.2{\cdot}\min\left(1, \frac{\text{numPairs}}{10}\right),$$

where higher values indicate more heterogeneous clusters that are therefore better candidates for subdivision.

Phase 3: Post adjustment toward the exact target
Finally, a **post adjustment** step enforces the *exact* target number of clusters. If there are too few clusters, we split those with the highest split priority; if there are too many, we merge clusters with the lowest priority. The result is a final set that respects the predefined rule budget.

Rule synthesis from clusters

Rule generation proceeds in the same way as in the K-Means generator: for each cluster we compute IP and port ranges and produce rules of the form `(srcIP, dstIP, srcPort, dstPort, protocol)`. Likewise, **controlled overlaps** can be introduced and configured by percentage (20% partial and 10% full in our case).

In this approach, **deduplication** is more critical than in K-Means because:

- hierarchical clustering can yield convergent clusters,
- the post adjustment step may produce similar sets,
- multiple subdivisions can lead to identical IP/port combinations,
- the overlaps themselves can introduce duplicates.

Duplicate rules (the same logical five tuple) are therefore **merged** by summing their `packet_count`, reducing cardinality without loss of information.

Advantages of the Hierarchical Top Down approach

This strategy provides precise control over granularity, reaching an exact target number of rules by splitting only those clusters that display genuine heterogeneity. It is computationally efficient, since a divide and conquer procedure lowers complexity and scales more effectively than flat clustering on large data sets. It is also highly interpretable: the hierarchy reflects networking semantics (for example, IP subnets and port groupings), and the split criteria have a clear meaning that preserves traceability of the resulting rules.

5.2.6 Comparison between the Two Rule Generators

The two approaches, **K-Means Rule Generator** and **Hierarchical Top-Down Rule Generator**, share the same logic of *adaptive feature engineering* on IP addresses and ports, but they differ substantially in clustering strategy and in how granularity is controlled.

The first approach (*K-Means*) performs **direct partitional clustering**, applying K-Means in a single step to IP+port features. It is characterised by simplicity, speed, and lower computational overhead. However, the resulting number of clusters is only *approximate* relative to the desired target, typically deviating by a few units, and quality depends on dimensionality and initialisation.

The second approach (*Hierarchical Top-Down*) implements **divisive hierarchical clustering** in three phases (IP $\rightarrow$ Ports $\rightarrow$ Post adjustment). It guarantees an *exact* match to the target number of rules and introduces fine grained control via split metrics (spread, range span, consecutiveness). This precision comes at the cost of greater implementation complexity and higher computational overhead.

In terms of application scenarios:

- The **K-Means Generator** is suitable for small to medium datasets, contexts with tight performance requirements, or cases in which an approximate number of rules is acceptable.
- The **Hierarchical Top-Down Generator** is preferable for large datasets and when thousands of rules are needed, or when a selected target of rules is strictly required.

In conclusion, the two generators should be viewed as **complementary**: the first prioritises efficiency and simplicity, whereas the second guarantees precision and the ability to adapt to complex scenarios. The choice between them thus depends on performance constraints and accuracy requirements in the deployment context. In our work, given a medium-small dataset (about 239,000 packets), we adopted the first generator to create the rule set.

It is important to emphasise that both generators require packets in the **five tuple** format `(srcIP, dstIP, srcPort, dstPort, protocol)`. To this end, from the *SCADA/ICS Network Captures* dataset we extracted a `.pcap` file, which was then transformed into the desired format by a Python function implemented with the `Scapy` library:

Listing 5.1. Extraction of 5-tuples from a SCADA/ICS PCAP file

```
from scapy.all import rdpcap, IP, TCP, UDP, ICMP
import csv

pcap_file = "4SICS-GeekLounge-151020.pcap"
packets = rdpcap(pcap_file)

with open("five_tuple_full.csv", "w", newline='') as csvfile:
    fieldnames = ['srcIP', 'dstIP', 'srcPort', 'dstPort', 'protocol']
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames, delimiter=';')
    writer.writeheader()
    for pkt in packets:
        if IP in pkt:
            src_ip, dst_ip, proto = pkt[IP].src, pkt[IP].dst, pkt[IP].proto
            if TCP in pkt:
                writer.writerow({'srcIP': src_ip, 'dstIP': dst_ip,
                                 'srcPort': pkt[TCP].sport, 'dstPort':
                                     pkt[TCP].dport,
                                 'protocol': 'TCP'})
            elif UDP in pkt:
                writer.writerow({'srcIP': src_ip, 'dstIP': dst_ip,
                                 'srcPort': pkt[UDP].sport, 'dstPort':
                                     pkt[UDP].dport,
                                 'protocol': 'UDP'})
            elif ICMP in pkt:
                writer.writerow({'srcIP': src_ip, 'dstIP': dst_ip,
                                 'srcPort': '', 'dstPort': '', 'protocol':
                                     'ICMP'})
            else:
                writer.writerow({'srcIP': src_ip, 'dstIP': dst_ip,
                                 'srcPort': '', 'dstPort': '', 'protocol':
                                     proto})
```

The result is a `CSV` file containing the entire traffic converted into the five tuple format.

Chapter 6

Implementation and Validation

In this section, we present the implementation and validation of our firewall models, focusing in particular on the **stateless firewall**, the state-based firewalls with **3, 9, and 27 states**, and the final **digital twin configuration**. The implementation was carried out in **Python**, a language widely adopted in the research community due to its readability, extensive ecosystem of scientific libraries, and strong support for machine learning and reinforcement learning. Python is particularly well suited to this work. First, it enables rapid prototyping: its readable, high level syntax supports fast experimentation and iteration. Second, it offers native support for reinforcement learning through libraries such as `Gymnasium` (a maintained fork of OpenAI Gym), which provide standard interfaces for building, training, and evaluating RL environments.

6.0.1 Reinforcement Learning Framework

We developed a reinforcement learning component using the **Gymnasium** library, because it offers a standard API for defining environments and agents, along with a diverse collection of reference environments. In particular a key feature of this framework is the use of **wrappers**, which enable the modification of existing environments without altering the underlying implementation. Wrappers simplify code maintenance, encourage modularity, and can be chained to combine multiple modifications.

The design of our smart firewall followed this paradigm: for each firewall type (stateless, state based, and digital twin), a custom wrapper was implemented, from which the actual RL environment was created. This modular approach allowed us to maintain a consistent structure across different configurations while adapting the environment to the specific requirements of each firewall type. In this way, every firewall variant can be described as a specialization of the same general framework, differing only in the way state information is represented.

6.1 Development of Baseline Firewall

The first implementation, referred to as the **baseline firewall**, introduces the stateless configuration of our system. The core environment is represented by the class `FirewallEnv`, which inherits from `gym.Env`. This class models the interaction between the agent (the smart firewall) and a stream of incoming packets, guided by a set of predefined filtering rules. The environment tracks the current packet, the actions attempted by the agent, and returns feedback in the form of rewards and penalties.

6.1.1 Environment Design

The environment is implemented as a wrapper around `gym.Env`. It exposes a single observation that reflects the stateless design, an action space that enumerates the firewall rules, and a reward signal that indicates whether the selected rule matches the current packet.

Formally:

- **States:** the observation is always 0, i.e. no contextual information is provided.
- **Actions:** $A = \{0, \ldots, |\mathcal{R}| - 1\}$, where $\mathcal{R}$ is the set of available rules.
- **Reward:**

$$r_t = \begin{cases} 0.77 & \text{if } \texttt{match}(p_t, r_a) = \text{True}, \\ 0 & \text{if } t < 30 \text{ and no match}, \\ -0.045 & \text{otherwise.} \end{cases}$$

- **Transition:** when a match occurs, the environment advances to the next packet; otherwise it remains on the same packet.

Listing 6.1. Stateless firewall environment

```
import gym
from gym import spaces

class FirewallEnv(gym.Env):
    def __init__(self, packets, rules, match_fn,
       neg_cost_after=30):
        super().__init__()
        self.packets = packets
        self.rules = rules
        self.match_fn = match_fn
        self.neg_cost_after = neg_cost_after

        self.current = 0
        self.observation_space = spaces.Discrete(1)
        self.action_space = spaces.Discrete(len(rules))
```

```
    def reset(self):
        self.current = 0
        return 0

    def step(self, action):
        if self.current >= len(self.packets):
            return 0, 0.0, True, {"miss": 0}

        packet = self.packets[self.current]
        rule = self.rules[action]
        is_match = self.match_fn(packet, rule)

        cast = 0.0 if self.current < self.neg_cost_after else
           -0.045
        reward = 0.77 if is_match else cast
        info = {"miss": 0 if is_match else 1}

        if is_match:
            self.current += 1
            done = (self.current >= len(self.packets))
        else:
            done = False

        return 0, reward, done, info
```

6.1.2 Action Selection Policy

The agent follows an ε-greedy policy. With probability ε, a random rule is chosen among those not yet attempted for the current packet (exploration). Otherwise, the agent selects the rule with the highest Q-value (exploitation).

Listing 6.2. Action selection with ε-greedy strategy

```
import numpy as np, random

def select_action(tried, q_table, epsilon, n_rules):
    available = [i for i in range(n_rules) if i not in tried]
    if not available:
        available = list(range(n_rules))

    if random.uniform(0, 1) < epsilon:
        return random.choice(available) # exploration
    else:
        q_vals = q_table[0, available]
        return available[np.argmax(q_vals)] # exploitation
```

6.1.3 Q-Learning Algorithm

The Q-learning update rule in this setting is simplified because there is only one state:

$$Q(a) \leftarrow Q(a) + \alpha\left(r - Q(a)\right),$$

which corresponds to a bandit problem with learning rate α and no discount factor ($\gamma = 0$).

The algorithm iterates over the packets, selecting actions until a match is found, at the end the total number of misses and the execution time are recorded and showed as performance metrics.

Listing 6.3. Q-learning procedure for the stateless firewall

```
import time
import numpy as np

def Q_learning(env, alpha, epsilon):
    n_rules = env.action_space.n
    q_table = np.zeros((1, n_rules))
    skip_counts = np.zeros(n_rules, dtype=int)

    miss_total = 0
    start = time.time()

    obs = env.reset()
    while env.current < len(env.packets):
        tried = []
        correct = False

        while not correct:
            action = select_action(tried, q_table, epsilon,
                n_rules)
            tried.append(action)
            _, reward, done, info = env.step(action)
            miss_total += info['miss']

            # Q-learning update
            q_table[0, action] += alpha * (reward - q_table[0,
                action])

            # Penalization for unused actions
            for i in range(n_rules):
                if i == action:
                    skip_counts[i] = 0
                else:
                    skip_counts[i] += 1
                    if skip_counts[i] >= 200:
                        q_table[0, i] -= 0.05 * abs(q_table[0, i])
```

```
                    skip_counts[i] = 0

            correct = reward > 0

    end = time.time()
    return miss_total, end - start
```

This baseline firewall demonstrates how reinforcement learning can be applied even in a stateless configuration. Despite its simplicity, the approach allows the agent to gradually learn an ordering of the rules that reduces the number of misses. However, the absence of state information limits the model's ability to capture contextual dependencies between packets. For this reason, subsequent versions introduce explicit states (3, 9, and 27), enabling more sophisticated learning strategies.

6.2 Development of 3 States Firewall

The second configuration extends the baseline by introducing an explicit **state space** with three discrete states that encode the transport protocol of the current packet: *TCP*, *UDP*, and *ICMP*. By conditioning the action-value function on protocol-specific states, the agent can learn different rule orderings for different classes of traffic. This richer representation improves the agent's ability to minimize *misses* and processing latency compared to the stateless baseline.

6.2.1 Environment Design

The environment takes from `gym.Env` and exposes:

- **State space:** $S = \{0,1,2\}$, in which $0 \to$ TCP, $1 \to$ UDP, $2 \to$ ICMP.
- **Action space:** $A = \{0, \ldots, |\mathcal{R}| - 1\}$, where $\mathcal{R}$ is the rule set.
- **Reward:** a positive base reward on a match and a negative base reward on a no-match, plus a small penalty proportional to the number of attempts on the same packet, to explicitly discourage the long search chains.

Formally, at time t the agent observes $s_t \in \{0,1,2\}$ and selects $a_t \in A$. Let $\texttt{match}(p_t, r_{a_t}) \in \{\text{True}, \text{False}\}$. The immediate reward is

$$r_t = \begin{cases} 1.0 + \underbrace{(-0.02)\, n_t}_{\text{attempt penalty}} & \text{if match,} \\ -0.1 + \underbrace{(-0.02)\, n_t}_{\text{attempt penalty}} & \text{otherwise,} \end{cases}$$

where n_t denotes the number of actions already attempted on the current packet before a_t. If a match occurs, the environment advances with the next packet (and thus to a new state consistent with its protocol); otherwise, the state remains unchanged.

Listing 6.4. Three-state firewall environment (TCP/UDP/ICMP)

```
import gym
from gym import spaces

class MultiStateFirewallEnv(gym.Env):
    def __init__(self, packets, rules, match_fn):
        super().__init__()
        self.packets = packets
        self.rules = rules
        self.match_fn = match_fn

        self.current = 0
        self.tried_actions = set()
        self.match_cache = {}

        # 3 states: 0=TCP, 1=UDP, 2=ICMP
        self.observation_space = spaces.Discrete(3)
        self.action_space = spaces.Discrete(len(rules))

        self.protocol_to_state = {'TCP': 0, 'UDP': 1, 'ICMP': 2}

    def get_state(self, packet):
        protocol = packet.get("protocol", "TCP").upper()
        return self.protocol_to_state.get(protocol, 0)

    def reset(self, seed=None):
        super().reset(seed=seed)
        self.current = 0
        self.tried_actions = set()
        self.match_cache = {}
        if len(self.packets) > 0:
            return self.get_state(self.packets[0]), {}
        return 0, {}

    def step(self, action):
        if self.current >= len(self.packets):
            return None, 0.0, True, {'miss': 0, 'attempts': 0,
                                     'packet_id': self.current,
                                        'state': None}

        packet = self.packets[self.current]
        rule = self.rules[action]

        cache_key = (self.current, action)
        if cache_key in self.match_cache:
            is_match = self.match_cache[cache_key]
        else:
```

```
            is_match = self.match_fn(packet, rule)
            self.match_cache[cache_key] = is_match

        # base reward + penalty for multiple attemps on the
           packet
        penalty = -0.02 * len(self.tried_actions) if
           self.tried_actions else 0.0
        base_reward = 1.0 if is_match else -0.1
        reward = base_reward + penalty

        info = {
            'miss': 0 if is_match else 1,
            'attempts': len(self.tried_actions) + 1,
            'packet_id': self.current,
            'state': self.get_state(packet)
        }

        if is_match:
            self.current += 1
            self.tried_actions = set()
            if self.current >= len(self.packets):
                return None, reward, True, info
            next_state = self.get_state(self.packets[self.current])
            return next_state, reward, False, info
        else:
            self.tried_actions.add(action)
            next_state = self.get_state(packet)
            return next_state, reward, False, info
```

6.2.2 Action Selection Policy

We employ an ε-greedy policy *conditioned on the current state*. The agent explores with probability ε among actions not yet tried for the current packet; otherwise, it exploits the action with maximum state-specific Q-value.

Listing 6.5. State-conditioned ε-greedy action selection

```
import numpy as np, random

def select_action_multistate(tried_actions, q_table, state,
   n_rules, epsilon):
    available = [i for i in range(n_rules) if i not in
       tried_actions]
    if not available: # fallback
        return random.randint(0, n_rules - 1)

    if random.random() < epsilon: # exploration
        return random.choice(available)
```

```
    else: # exploitation
        q_vals = [q_table[state][a] for a in available]
        return available[int(np.argmax(q_vals))]
```

6.2.3 Q-Learning Algorithm

With multiple states, the agent learns a separate action-value function for each protocol state. We use the standard tabular Q-learning update

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big],$$

with learning rate α, discount factor $\gamma = 0.9$, and a multiplicative decay on ε to anneal exploration. We also keep per-state statistics (packets, misses) for monitoring.

Listing 6.6. Q-learning for the three-state firewall

```
import time
import numpy as np

def Q_learning_multistate(env, alpha=0.01, epsilon=0.1,
    decay_rate=0.995, gamma=0.9, verbose=False):
    n_rules = env.action_space.n
    q_table = np.zeros((env.observation_space.n, n_rules),
        dtype=float)

    miss_total = 0
    state_stats = {0: {'name': 'TCP', 'packets': 0, 'misses': 0},
                   1: {'name': 'UDP', 'packets': 0, 'misses': 0},
                   2: {'name': 'ICMP', 'packets': 0, 'misses': 0}}

    start = time.time()
    state, _ = env.reset()

    while env.current < len(env.packets):
        packet = env.packets[env.current]
        s = env.get_state(packet)
        state_stats[s]['packets'] += 1

        correct = False
        while not correct:
            a = select_action_multistate(env.tried_actions,
                q_table, s, n_rules, epsilon)

            old_q = q_table[s, a]
            next_state, reward, done, info = env.step(a)

            if next_state is not None:
```

```
                max_next_q = np.max(q_table[next_state])
            else:
                max_next_q = 0.0

            q_table[s, a] = old_q + alpha * (reward + gamma *
                max_next_q - old_q)

            miss_total += info['miss']
            if info['miss'] == 1:
                state_stats[s]['misses'] += 1

            correct = (reward > 0.0)

        epsilon = max(0.001, epsilon * decay_rate)

    elapsed = time.time() - start
    return miss_total, elapsed, q_table, state_stats
```

Compared to the stateless baseline, the 3 states model leverages protocol-aware conditioning to learn distinct orderings of rules per traffic class. The attempt penalty explicitly favors faster matches, thereby reducing per-packet search depth. Empirically, this results in fewer *misses* and lower latency, while preserving a compact tabular representation and a simple training loop. This design also serves as a stepping stone toward higher-cardinality state spaces (e.g., 9 and 27 states), where additional contextual features are encoded.

6.3 Development of 9 states firewall

In this configuration the state representation is composed by combining **transport protocol** (TCP/UDP/ICMP) with the **source port category** (system, user, dynamic). This result in a 3×3 grid that yields a discrete state space with nine elements. Thanks to this configuration the agent can specialize rule orderings for finer grained traffic patterns, improving match efficiency and latency.

6.3.1 Environment Design

The environment exposes:

- **State space:** $S = \{0, \ldots, 8\}$ obtained via

$$s(p) \;=\; \underbrace{\texttt{base}(\text{protocol})}_{\in\{0,1,2\}} \cdot 3 \;+\; \underbrace{\texttt{bucket}(\text{srcPort})}_{\in\{0,1,2\}},$$

  where $\texttt{base}(\cdot)$ maps TCP/UDP/ICMP to $\{0,1,2\}$, and $\texttt{bucket}(\cdot)$ bins the source port into

$$[0{,}1023] \to 0, \quad [1024{,}49151] \to 1, \quad [49152{,}65535] \to 2.$$

- **Action space:** $A = \{0, \ldots, |\mathcal{R}| - 1\}$, index of firewall rules.
- **Reward:** same as the 3 states one.

Listing 6.7. Nine-state firewall environment (protocol × source-port tier)

```
import gym
from gym import spaces

class MultiStateFirewallEnv9(gym.Env):
    def __init__(self, packets, rules, match_fn):
        super().__init__()
        self.packets = packets
        self.rules = rules
        self.match_fn = match_fn

        self.current = 0
        self.tried_actions = set()
        self.match_cache = {}

        # 9 states: 3 protocols x 3 port tiers
        self.observation_space = spaces.Discrete(9)
        self.action_space = spaces.Discrete(len(rules))

        self.protocol_to_base = {'TCP': 0, 'UDP': 1, 'ICMP': 2}
        self.port_ranges = [(0,1023), (1024,49151), (49152,65535)]

    def get_port_category(self, port):
        for i, (lo, hi) in enumerate(self.port_ranges):
            if lo <= port <= hi:
                return i
        return 1 # default: user/registered

    def get_state(self, packet):
        protocol = packet.get("protocol", "TCP").upper()
        src_port = packet.get("srcPort", 1024)
        base = self.protocol_to_base.get(protocol, 0)
        tier = self.get_port_category(src_port)
        return base * 3 + tier # in [0..8]

    def get_state_name(self, state_id):
        prot = ['TCP','UDP','ICMP'][state_id // 3]
        tier = ['System','User','Dynamic'][state_id % 3]
        return f"{prot}-{tier}"

    def reset(self, seed=None):
        super().reset(seed=seed)
        self.current = 0
        self.tried_actions = set()
```

```
        self.match_cache = {}
        if self.packets:
            return self.get_state(self.packets[0]), {}
        return 0, {}

    def step(self, action):
        if self.current >= len(self.packets):
            return None, 0.0, True, {'miss': 0, 'attempts': 0,
                                     'packet_id': self.current,
                                        'state': None}

        packet = self.packets[self.current]
        rule = self.rules[action]

        key = (self.current, action)
        if key in self.match_cache:
            is_match = self.match_cache[key]
        else:
            is_match = self.match_fn(packet, rule)
            self.match_cache[key] = is_match

        penalty = -0.02 * len(self.tried_actions) if
           self.tried_actions else 0.0
        reward = (1.0 if is_match else -0.1) + penalty

        info = {'miss': 0 if is_match else 1,
                'attempts': len(self.tried_actions) + 1,
                'packet_id': self.current,
                'state': self.get_state(packet)}

        if is_match:
            self.current += 1
            self.tried_actions = set()
            if self.current >= len(self.packets):
                return None, reward, True, info
            next_state = self.get_state(self.packets[self.current])
            return next_state, reward, False, info
        else:
            self.tried_actions.add(action)
            next_state = self.get_state(packet) # unchanged
            return next_state, reward, False, info
```

6.3.2 Action Selection Policy

We reuse the state-conditioned ε-greedy policy introduced before (Listing 6.5). The action-selection function is equal as before, but it's applied with the 9 states observation space.

6.3.3 Q-Learning Algorithm

We learn a tabular action-value function with shape $9 \times |\mathcal{R}|$ using the standard update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big]$$

It contains a learning rate α,a discount factor $\gamma = 0.9$, and a multiplicative decay of ε to anneal exploration. Moreover we keep the count for packets misses to analyze the performances.

Listing 6.8. Q-learning for the nine-state firewall

```
import time
import numpy as np

def Q_learning_multistate9(env, alpha=0.01, epsilon=0.1,
   decay_rate=0.995, gamma=0.9, verbose=False):
    n_states = env.observation_space.n # 9
    n_rules = env.action_space.n
    q_table = np.zeros((n_states, n_rules), dtype=float)

    miss_total = 0
    state_stats = {i: {'name': env.get_state_name(i), 'packets':
       0, 'misses': 0}
                for i in range(n_states)}

    start = time.time()
    state, _ = env.reset()

    while env.current < len(env.packets):
        packet = env.packets[env.current]
        s = env.get_state(packet)
        state_stats[s]['packets'] += 1

        correct = False
        while not correct:
            a = select_action_multistate(env.tried_actions,
               q_table, s, n_rules, epsilon)

            old_q = q_table[s, a]
            next_state, reward, done, info = env.step(a)

            max_next_q = 0.0 if next_state is None else
               float(np.max(q_table[next_state]))
            q_table[s, a] = old_q + alpha * (reward + gamma *
               max_next_q - old_q)

            miss_total += info['miss']
```

```
            if info['miss'] == 1:
                state_stats[s]['misses'] += 1

            correct = (reward > 0.0)

        epsilon = max(0.001, epsilon * decay_rate)

    elapsed = time.time() - start

    return miss_total, elapsed, q_table, state_stats
```

Against the 3 states model, the 9 states representation offers a more nuanced conditioning by incorporating port tiers, which are often indicative of service classes and traffic behavior. The attempt penalty continues to steer the agent toward shallow match paths. While the larger state space increases fragmentation of experience, the tabular setting remains tractable, and the per-state statistics provide visibility into which protocol/port combinations benefit most from specialized rule orderings.

6.4 Development of 27 states firewall

This configuration further enriches the state representation by combining **transport protocol** (TCP/UDP/ICMP) with both the **source** and **destination port tiers** (system, user, dynamic). The resulting grid $3 \times 3 \times 3$ yields a discrete state space with 27 elements, enabling the agent to specialize rule orderings for specific protocol/port patterns and to shorten match chains.

6.4.1 Environment Design

The environment exposes:

- **State space:** $S = \{0, \ldots, 26\}$ computed as

$$s(p) \;=\; \underbrace{\texttt{base}(\text{protocol})}_{\in\{0,1,2\}} \cdot 9 \;+\; \underbrace{\texttt{bucket}(\text{srcPort})}_{\in\{0,1,2\}} \cdot 3 \;+\; \underbrace{\texttt{bucket}(\text{dstPort})}_{\in\{0,1,2\}},$$

  where $\texttt{bucket}(\cdot)$ maps port ranges $[0,1023] \to 0$, $[1024,49151] \to 1$, $[49152,65535] \to 2$.

- **Action space:** $A = \{0, \ldots, |\mathcal{R}| - 1\}$, the rule indices.

- **Reward:** same as the 3 states one.

Listing 6.9. Twenty-seven-state firewall environment (protocol × src-port tier × dst-port tier)

```
import gym
from gym import spaces
```

```
class MultiStateFirewallEnv27(gym.Env):
    def __init__(self, packets, rules, match_fn):
        super().__init__()
        self.packets = packets
        self.rules = rules
        self.match_fn = match_fn

        self.current = 0
        self.tried_actions = set()
        self.match_cache = {}

        # 27 states: 3 (protocol) x 3 (src tier) x 3 (dst tier)
        self.observation_space = spaces.Discrete(27)
        self.action_space = spaces.Discrete(len(rules))

        self.protocol_to_base = {'TCP': 0, 'UDP': 1, 'ICMP': 2}
        self.port_ranges = [(0,1023), (1024,49151), (49152,65535)]
        self.port_names = ['System', 'User', 'Dynamic']

    def get_port_category(self, port):
        for i, (lo, hi) in enumerate(self.port_ranges):
            if lo <= port <= hi:
                return i
        return 1 # default: user/registered

    def get_state(self, packet):
        protocol = packet.get("protocol", "TCP").upper()
        src_port = packet.get("srcPort", 1024)
        dst_port = packet.get("dstPort", 80)

        P = self.protocol_to_base.get(protocol, 0)
        S = self.get_port_category(src_port)
        D = self.get_port_category(dst_port)
        return P * 9 + S * 3 + D # in [0..26]

    def get_state_name(self, state_id):
        prot = ['TCP','UDP','ICMP'][state_id // 9]
        rem = state_id % 9
        s_t = rem // 3
        d_t = rem % 3
        return f"{prot}-Src{self.port_names[s_t]}" +
                f"-Dst{self.port_names[d_t]}"

    def reset(self, seed=None):
        super().reset(seed=seed)
        self.current = 0
        self.tried_actions = set()
        self.match_cache = {}
```

```
        if self.packets:
            return self.get_state(self.packets[0]), {}
        return 0, {}

    def step(self, action):
        if self.current >= len(self.packets):
            return None, 0.0, True, {'miss': 0, 'attempts': 0,
                                     'packet_id': self.current,
                                        'state': None}

        packet = self.packets[self.current]
        rule = self.rules[action]

        key = (self.current, action)
        if key in self.match_cache:
            is_match = self.match_cache[key]
        else:
            is_match = self.match_fn(packet, rule)
            self.match_cache[key] = is_match

        penalty = -0.02 * len(self.tried_actions) if
           self.tried_actions else 0.0
        reward = (1.0 if is_match else -0.1) + penalty

        info = {'miss': 0 if is_match else 1,
                'attempts': len(self.tried_actions) + 1,
                'packet_id': self.current,
                'state': self.get_state(packet)}

        if is_match:
            self.current += 1
            self.tried_actions = set()
            if self.current >= len(self.packets):
                return None, reward, True, info
            next_state = self.get_state(self.packets[self.current])
            return next_state, reward, False, info
        else:
            self.tried_actions.add(action)
            next_state = self.get_state(packet)
            return next_state, reward, False, info
```

6.4.2 Action Selection Policy

We reuse the state-conditioned ε-greedy policy introduced before (Listing 6.5). The action-selection function is equal as before, but it's applied with the 27 states observation space.

6.4.3 Q-Learning Algorithm

We learn a tabular action-value function with shape $27 \times |\mathcal{R}|$ using the standard update:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \Big[r_t + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t) \Big]$$

It contains a learning rate α,a discount factor $\gamma = 0.9$, and a multiplicative decay of ε to anneal exploration. Moreover we keep the count for packets misses to analyze the performances.

Listing 6.10. Q-learning for the twenty-seven-state firewall

```
import time
import numpy as np

def Q_learning_multistate27(env, alpha=0.01, epsilon=0.1,
   decay_rate=0.995, gamma=0.9, verbose=False):
    n_states = env.observation_space.n # 27
    n_rules = env.action_space.n
    q_table = np.zeros((n_states, n_rules), dtype=float)

    miss_total = 0
    state_stats = {i: {'name': env.get_state_name(i), 'packets':
       0, 'misses': 0}
                for i in range(n_states)}

    start = time.time()
    state, _ = env.reset()

    while env.current < len(env.packets):
        packet = env.packets[env.current]
        s = env.get_state(packet)
        state_stats[s]['packets'] += 1

        correct = False
        while not correct:
            a = select_action_multistate(env.tried_actions,
               q_table, s, n_rules, epsilon)

            old_q = q_table[s, a]
            next_state, reward, done, info = env.step(a)

            max_next_q = 0.0 if next_state is None else
               float(np.max(q_table[next_state]))
            q_table[s, a] = old_q + alpha * (reward + gamma *
               max_next_q - old_q)

            miss_total += info['miss']
```

```
            if info['miss'] == 1:
                state_stats[s]['misses'] += 1

            correct = (reward > 0.0)

        epsilon = max(0.001, epsilon * decay_rate)

    elapsed = time.time() - start

    return miss_total, elapsed, q_table, state_stats
```

The 27 states representation captures both endpoints service tiers in addition to the protocol, allowing the agent to specialize rule orderings for bidirectional traffic profiles. The attempt penalty continues to bias toward shallow search paths, reducing latency. While the larger state space increases data fragmentation and may slow convergence, the tabular approach remains tractable for the evaluated rule set sizes, and per-state statistics provide actionable diagnostics on where specialization yields the largest gains.

6.5 Development of the Digital Twin Configuration

We introduce a *digital twin* (two-stage) architecture that combines a **Traditional Firewall** with an **RL agent**. We experimented with three agent variants:

1. **Baseline Simple Configuration** (stateless, fixed hyperparameters);
2. **Enhanced Simple Configuration** (stateless, adaptive hyperparameters);
3. **State-Aware Configuration** (27 states).

The first and the third correspond to the smart firewall agents described previously. The traditional firewall performs a *first matching rule* policy with a rule order that is periodically updated from the agent's learned *Q-values*. In particular, the two components process the same traffic in parallel, the same packet is passed to both components; the rule table is updated by swapping an ordered view.

6.5.1 General Architecture

The traditional firewall maintains a rule list and applies first-match semantics. The method `update_rule_order` consumes the agent's Q-table to sort the rules in descending utility, while `process_packet` evaluates the current ordered list.

Listing 6.11. Traditional firewall with first-matching rule

```
import time
```

```
class TraditionalFirewall:
    def __init__(self, rules):
        self.rules = rules
        self.ordered_rules = list(range(len(rules)))
        self.packet_count = 0
        self.total_misses = 0
        self.processing_time = 0.0

    def update_rule_order(self, q_table):
        """Sort rules by descending Q-value produced by the
           agent."""
        q_values = q_table[0]
        self.ordered_rules = sorted(
            range(len(self.rules)),
            key=lambda i: q_values[i],
            reverse=True
        )

    def process_packet(self, packet):
        """Apply first-matching rule on the current ordered
           list."""
        start = time.perf_counter()
        self.packet_count += 1
        misses = 0
        for idx in self.ordered_rules:
            rule = self.rules[idx]
            if match(packet, rule):
                break
            misses += 1
        self.processing_time += (time.perf_counter() - start)
        self.total_misses += misses
        return misses
```

The `DualFirewallSystem` wires the agent and the traditional component. The agent exposes a Q-table and an `update(packet)` routine (training step on the fly). The coordinator feeds both components the same packet stream and periodically refreshes the traditional ordering.

Listing 6.12. Dual firewall coordinator

```
class DualFirewallSystem:
    def __init__(self, packets, rules, agent,
                 update_interval=20000):
        """
        packets: iterable of packets
        rules: rule set shared by both components
        agent: exposes agent.q_table and agent.update(packet)
```

```
        """
        self.packets = packets
        self.traditional_fw = TraditionalFirewall(rules)
        self.agent = agent
        self.update_interval = update_interval

        # Initial ordering from the (possibly warm-started) agent
        self.traditional_fw.update_rule_order(self.agent.q_table)

    def run(self):
        """
        Sequential coordination (evaluation-friendly).
        """
        for i, pkt in enumerate(self.packets, start=1):
            # Agent learns/refreshes Q-values on the fly
            self.agent.update(pkt)

            # Traditional FW evaluates with current ordering
            self.traditional_fw.process_packet(pkt)

            # Periodically refresh ordering from the agent's
               Q-table
            if i % self.update_interval == 0:
                self.traditional_fw.
                    update_rule_order(self.agent.q_table)
```

The listing adopts a sequential loop to make the control flow explicit. In a parallel deployment, the agent updates its Q-table on a shadow copy while the traditional firewall keeps processing; at each `update_interval`, the coordinator performs an atomic swap of the ordered view. This realizes the digital-twin behavior: wall-clock time is dominated by the slower of the two tasks (processing vs. update), while maintaining first-match semantics with an agent-driven, continually refreshed rule order.

6.5.2 Enhanced Strategy

The *Enhanced Baseline* augments the stateless tabular Q-learning agent with richer reward shaping, adaptive exploration and learning rates, and lateral credit assignment across related rules, complemented by lightweight per-rule telemetry to guide online learning. The objective is to shorten match chains, prioritize consistently effective rules, and remain responsive when performance deteriorates.

Implementation Design

The agent has a single row Q-table, corresponding to a single symbolic observation. For each packet selects an action using an adaptive ε, computes an adaptive reward that reflects both the number of attempts and the rank of the eventual match,

updates the selected Q-value with the learning rate α, and diffuses a small bonus to rules considered similar (approximated here by adjacent indices).

Listing 6.13. Enhanced Baseline

```
from collections import deque
import numpy as np
import random

class ImprovedQLearningFirewall:
    def __init__(self, dataset_packets, rules):
        self.dataset_packets = dataset_packets
        self.rules = rules
        self.q_table = np.zeros((1, len(rules)))

        # Per-rule telemetry (drives adaptivity)
        self.rule_stats = {
            'success_count': np.zeros(len(rules)),
            'attempt_count': np.zeros(len(rules)),
            'avg_position': np.zeros(len(rules)), # average rank
               when successful
            'recent_success_rate':np.zeros(len(rules))
        }
        self.recent_performance = deque(maxlen=1000) # last-N
           packets (tries, etc.)

    def select_action(self, tried, epsilon):
        avail = [i for i in range(len(self.rules)) if i not in
           tried] or list(range(len(self.rules)))
        if random.random() < epsilon:
            return random.choice(avail)
        q_vals = self.q_table[0, avail]
        return avail[int(np.argmax(q_vals))]

    def get_adaptive_epsilon(self, base_epsilon):
        """Exploration adapts to recent average tries (worse ->
           explore more)."""
        if len(self.recent_performance) < 100:
            return base_epsilon
        recent_avg_tries = np.mean([p['tries'] for p in
           list(self.recent_performance)[-100:]])
        return min(0.3, base_epsilon * 1.5) if recent_avg_tries >
           3 else max(0.01, base_epsilon * 0.7)

    def get_adaptive_alpha(self, action):
        """Per-action learning rate: higher when the rule is less
           explored."""
        attempts = self.rule_stats['attempt_count'][action]
        if attempts < 50: return 0.15
```

```
    if attempts < 200: return 0.12
    return 0.08

def calculate_adaptive_reward(self, action, is_match,
   position_in_tries, tried_actions):
    """
    Reward shaping:
      - incremental penalty per attempt on the same packet
      - positive floor for matches
      - speed bonus for early matches
      - failure penalty proportional to the rule's historical
         failure rate
    """
    penalty = -0.002 * max(0, len(tried_actions)) # attempt
       cost
    base = 1.1 if is_match else -0.38

    if is_match:
        reward = max(0.1, base + penalty) # positive floor
        speed_bonus = max(0.0, 0.2 - 0.05 * position_in_tries)
           # earlier is better
        reward += speed_bonus
    else:
        reward = base + penalty
        fail_rate = 1.0 - (
           self.rule_stats['success_count'][action] /
           max(1, self.rule_stats['attempt_count'][action]) )
        reward += -0.1 * fail_rate # penalize unreliable rules
    return reward

def update_similar_rules(self, successful_action,
   bonus_reward):
    for i in range(max(0, successful_action - 2),
       min(len(self.rules), successful_action + 3)):
        if i != successful_action:
            self.q_table[0, i] += 0.01 * bonus_reward

def process_and_update(self, packet, base_epsilon=0.1):
    tried, position, correct = [], 0, False
    while not correct:
        position += 1
        eps = self.get_adaptive_epsilon(base_epsilon)
        action = self.select_action(tried, eps)
        tried.append(action)

        # environment check (omitted here): is_match =
           match(packet, self.rules[action])
        is_match = match(packet, self.rules[action])
```

```
            self.update_rule_statistics(action, is_match, position)
            reward = self.calculate_adaptive_reward(action,
                is_match, position, tried)
            alpha = self.get_adaptive_alpha(action)

            self.q_table[0, action] += alpha * (reward -
                self.q_table[0, action])
            if is_match:
                self.update_similar_rules(action, 0.1 * reward)
            correct = is_match

        self.recent_performance.append({'tries': position})
```

Comparison with the Baseline Simple agent

Compared with a simple stateless agent that keeps ε and α fixed, assigns a constant reward to correct matches, and applies penalties that rise in steps, the enhanced design introduces four coordinated changes that shorten decision paths and improve data efficiency. First, the reward is reshaped: each additional attempt carries an extra cost, successful outcomes receive a guaranteed minimum positive return, and an additional speed bonus is granted that diminishes as the successful rule appears later in the attempt sequence. Second, exploration adapts to recent performance: the rate ε is raised when the average number of attempts per packet worsens and lowered when it improves, producing a balance between exploration and exploitation that adjusts itself. Third, the learning rate α follows a coarse schedule that accelerates learning in the initial phase and stabilises updates thereafter, without the need for per rule counters. Finally, credit is propagated laterally: a small portion of the reward diffuses to neighbouring rules as a stand in for rule similarity, which increases sample efficiency when adjacent rules share structure, for example along protocol or port ranges.

These modifications preserve the simplicity of tabular Q-learning while providing meaningful inductive biases (short match chains, responsiveness to drift, and mild correlation sharing). In practice, the enhanced agent converges to high-quality orderings faster than the baseline and maintains stable performance under varying traffic mixes.

6.6 Evaluation Protocol

All experiments were executed on **Google Colab**, a hosted Jupyter environment that requires no local setup and offers on-demand CPU runtimes (with optional GPU/TPU accelerators, not used here). In our CPU only sessions the typical profile exposed about *2 virtual CPUs* and *12–13 GB* of RAM (occasionally a high-RAM profile of ∼25 GB). As Colab resources are ephemeral and not guaranteed, with session timeouts and possible hardware variability, we structured each experiment

as a self contained run and fixed random seeds to preserve reproducibility across sessions.

As introduced in Chapter 5, we used the *SCADA/ICS Network Captures* dataset. Via the rule generator we derived three rule sets with **100**, **200**, and **500** rules, which were used for validation. To emulate recurring traffic, we adopted a two phase procedure: first, a pre–training of **3 epochs** over the ∼239k packet corpus (≈ 717k packets in total), then a validation on **1,000,000** packets obtained by repeating the dataset. We first assess the behavior of the *single smart firewalls* (across agent variants) and, in parallel, evaluate a *digital twin* configuration; finally, we compare the two paradigms on the same protocol and metrics to characterize their respective trade offs.

6.6.1 Single Smart Firewall Results

In this section we evaluate the *single smart firewall* by focusing on two primary metrics: the **total number of misses**, defined as the cumulative count of rule checks that fail before the first matching rule is found (first match policy); and the **processing time**, measured as the wall–clock time required to process the entire validation stream on a CPU only runtime.

Stateless vs State-Aware: comparative results

We compare the stateless baseline against state-aware agents with 3, 9, and 27 states (Figure 6.1 and 6.2), using 100 rules. Results clearly indicate that introducing state information substantially improves both accuracy and latency. The stateless model attains **8,690,541** total misses and **417.19 s** of validation time. Moving to 3 and 9 states reduces misses by roughly **86–87%** (to **1,147,403** and **1,093,380**, respectively), and halves the processing time (**207.60 s** and **204.04 s**). The 27 states configuration yields the best performance, with **486,965** total misses (**94.4% fewer than baseline**) and **149.13,s** total time (**64.3% lower than baseline**).

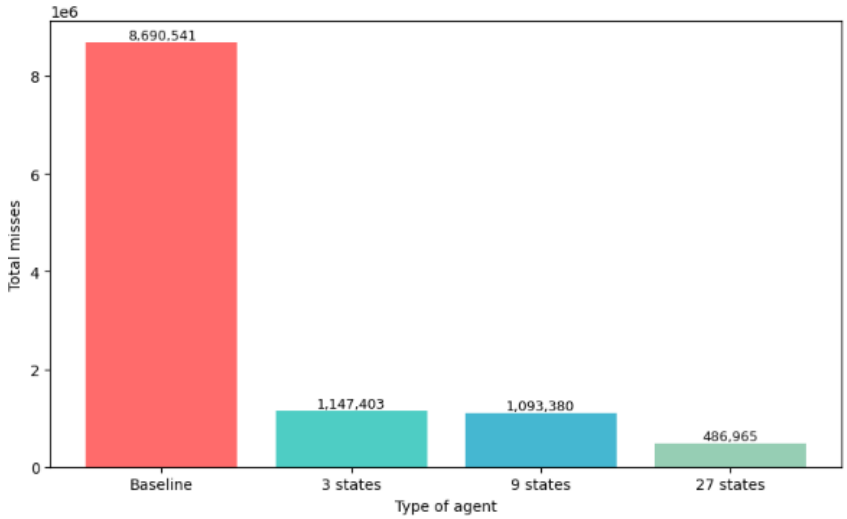


Figure 6.1. Smart Firewall: Total Misses

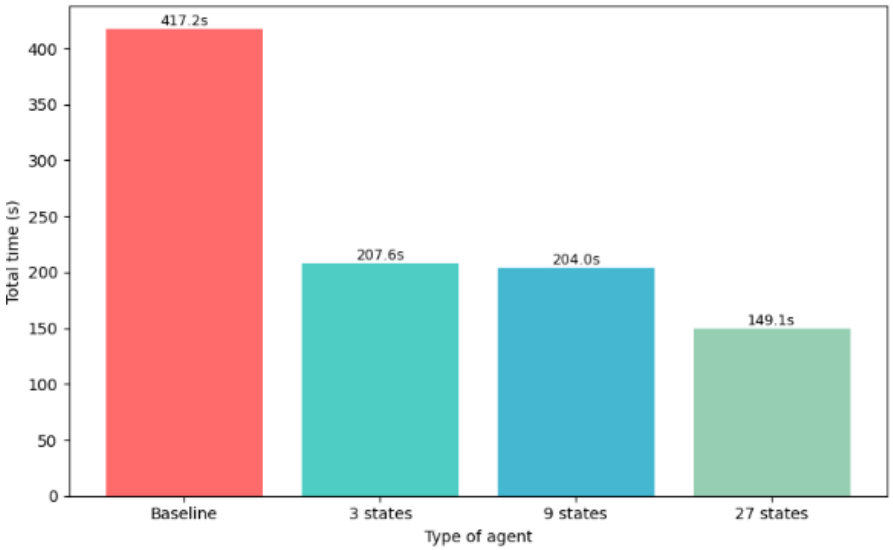


Figure 6.2. Smart Firewall: Total Time

These trends confirm that coarse grained protocol/port context enables the learner to concentrate probability mass on effective rules earlier in the sequence, shortening match chains and lowering wall clock time.

In light of these results, we discard the stateless baseline: its accuracy and latency are markedly inferior. We therefore concentrate on the three state-aware agents and study their scalability as the rule set grows.

Scalability with Larger Rule Sets

We now assess how state-aware agents scale as the rule set grows, focusing on *total misses* and *total validation time*. The analysis considers two larger configurations and contrasts 3, 9, and 27 state models.

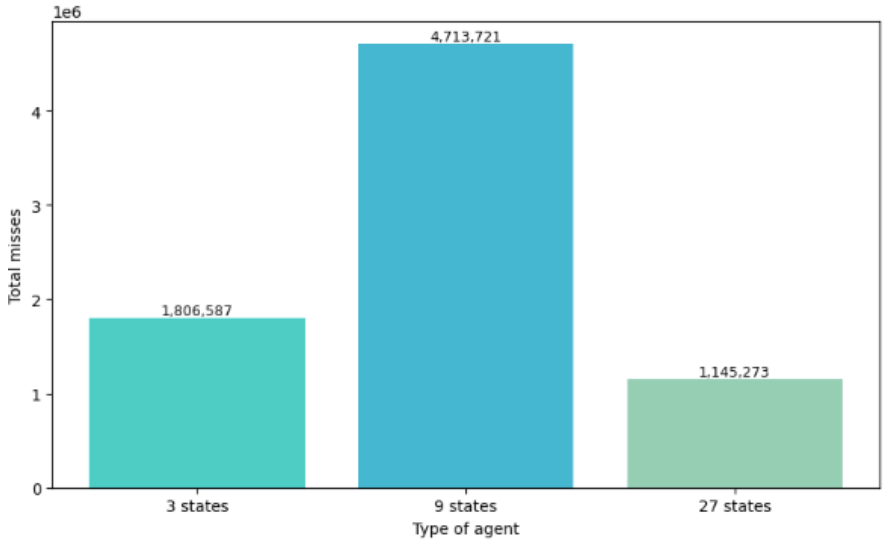


Figure 6.3. State-Aware Firewall: Total Misses with 200 rules

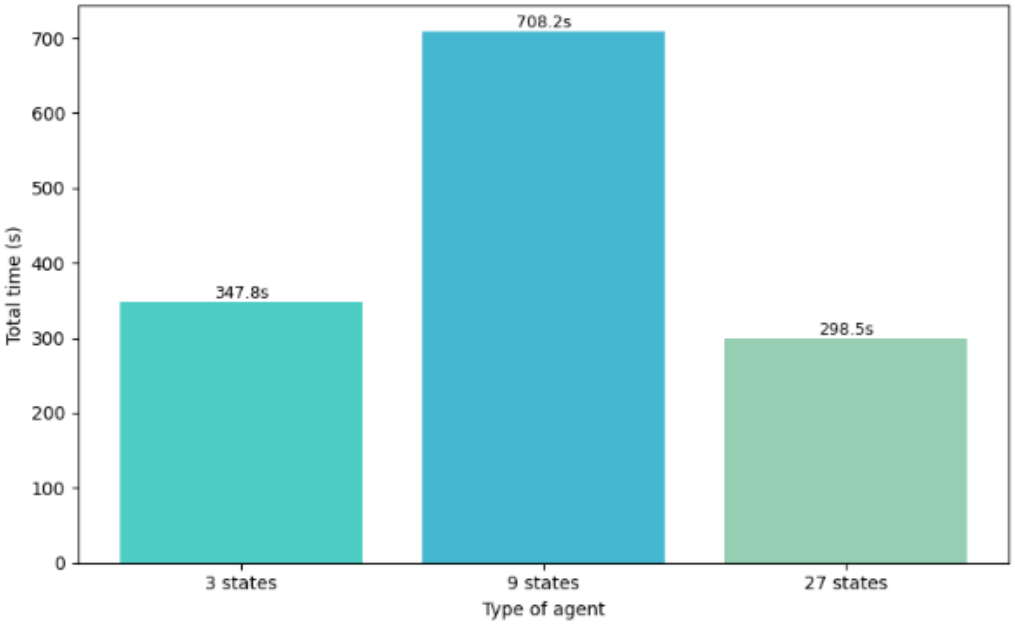


Figure 6.4. State-Aware Firewall: Total Time with 200 rules

With 200 rules (Figure 6.3 and 6.4), the 27 states agent achieves the best trade-off, with **1,145,273** total misses and **298.47 s** total time, outperforming both the 3 states (**1,806,587** misses; **347.81 s**) and the 9 states model (**4,713,721** misses; **708.25 s**). Relative to 3 states, the 27 state configuration reduces misses by approximately **36.6%** and time by **14.2%**; versus 9 states, the reductions are about **75.7%** (misses) and **57.9%** (time). The 9 states variant underperforms, suggesting that the intermediate partitioning may dilute learning signal across state–action pairs without delivering enough contextual gain.

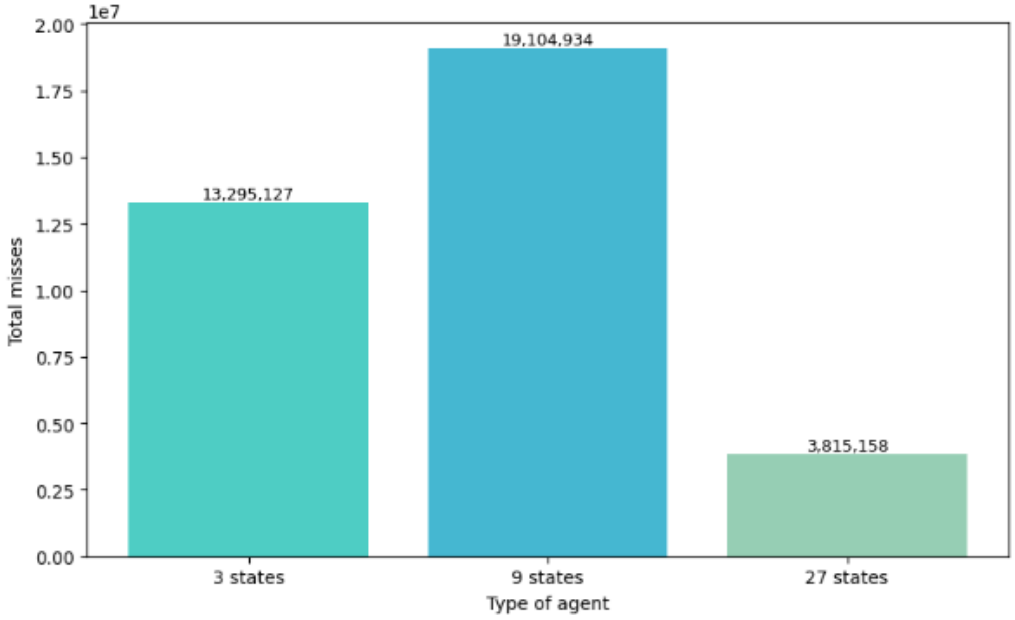


Figure 6.5. State-Aware Firewall: Total Misses with 500 rules

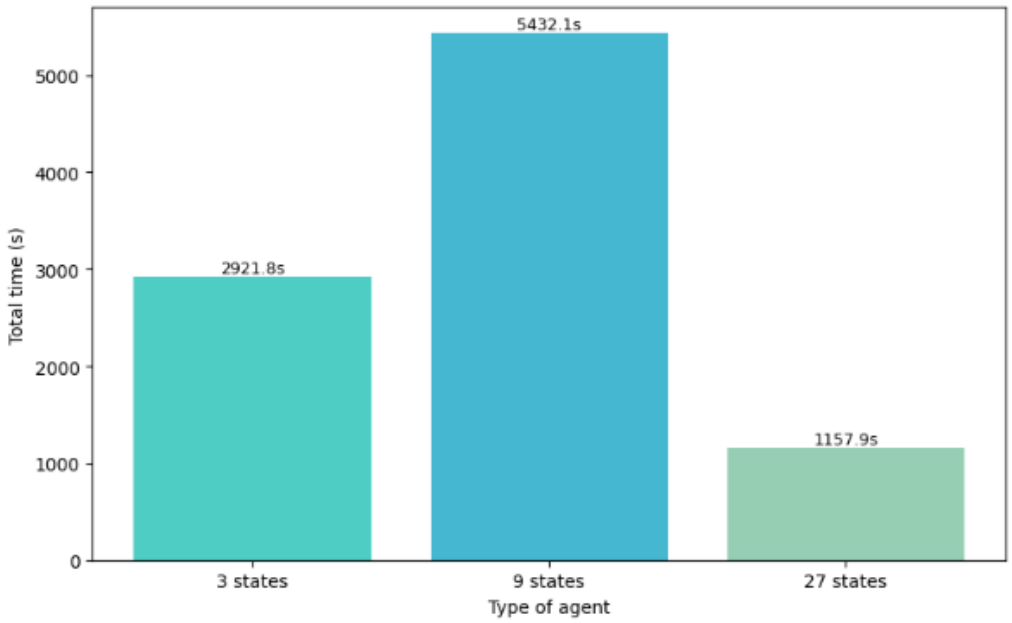


Figure 6.6. State-Aware Firewall: Total Time with 500 rules

At 500 rules, the gaps widen (Figure 6.5 and 6.6). The 27 states agent records **3,815,158** misses and **1,157.93 s**, while the 3 states and 9 states models reach **13,295,127** / **2,921.83 s** and **19,104,934** / **5,432.13 s**, respectively. Against 3 states, the 27 states model cuts misses by roughly **71.3%** and time by **60.3%**; against 9 states, improvements are about **80.0%** (misses) and **78.7%** (time). These results indicate that finer-grained state information becomes increasingly beneficial as the rule space expands.

Across both scales, the **27 states** agent consistently dominates, with substantial gains in accuracy (fewer misses) and latency (lower total time). The **3 state** model remains competitive at moderate sizes but loses ground as complexity grows. The **9 states** configuration is consistently inferior on this dataset, likely due to over-partitioning that weakens value estimates. Overall, richer state representations yield more efficient rule ordering under first-match semantics, and their advantage compounds with larger rule sets.

6.6.2 Digital Twin Results

We now assess the RL agents from Chapter 5 in a *digital twin* configuration. The evaluation targets three quantities: the **total number of misses** (first-match policy on the traditional firewall), the **processing time** of the traditional firewall over the validation stream, and the **agent-side update time**, i.e., the time the learning agent spends processing the same packets and updating the rule order. Although these timings are recorded separately for verification and analysis, in deployment the two components operate *in parallel*; consequently, the effective wall-clock per update window is dominated by the *critical path*, approximately $\max\{T_{\text{proc}}, T_{\text{agent}}\}$. We also study the impact of the **update interval**, the

number of packets between consecutive reorderings of the rule table, considering {1000, 5000, 10000, 20000, 50000} packets. This setup allows us to observe how accuracy (misses) and latency (processing vs. update time) co-evolve as the refresh frequency changes.

From the *total misses* plot (Figure 6.7) we observe a clear monotonic trend: the number of misses increases as the **update interval** grows. The reason is structural: between two reorderings the traditional firewall operates with a fixed rule table (first–match policy). When traffic characteristics drift, a large interval leaves the table *stale* for longer, so packets are evaluated against a suboptimal ordering, lengthening the search chain and yielding more misses. The ranking is stable across all intervals: **State-Aware (27 states)** attains the fewest misses, **Enhanced** is intermediate, and the **Baseline** is worst. Concretely, at **1,000** packets the 27 states model records ∼2.0 M misses versus ∼3.3 M (Enhanced) and ∼3.9 M (Baseline), i.e., a reduction of about **39**% and **49**%, respectively. At **50,000** packets, the 27 states model reaches ∼17.2 M misses, still below Enhanced (∼19.7 M) and well below Baseline (∼27.0 M), corresponding to improvements of approximately **13**% (vs Enhanced) and **36**% (vs Baseline). Overall, richer state information consistently mitigates staleness by concentrating probability mass on effective rules earlier in the sequence.

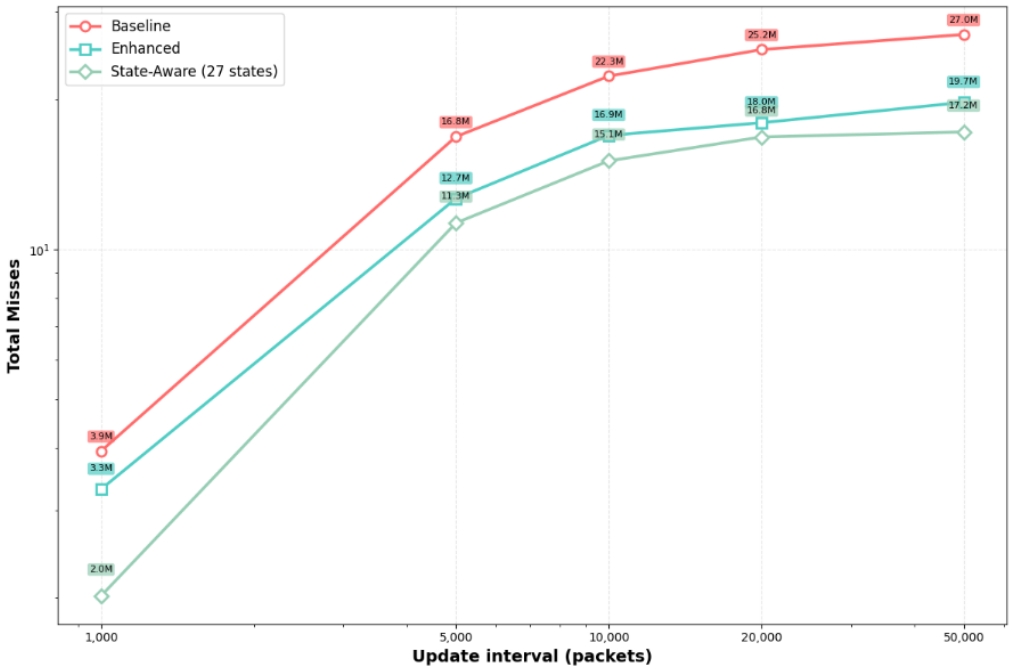


Figure 6.7. Comparation of total misses of RL agents in a digital twin configuration

The *processing time* exhibits the same monotonic trend (Figure 6.8): it increases with the **update interval**. This is consistent with the miss dynamics under first–match semantics; more misses imply longer rule chains per packet and therefore higher wall–clock time. The relative ranking mirrors the accuracy plot: **State-Aware (27 states)** is fastest, **Enhanced** is close behind, and **Baseline** is the slowest. Concretely, at **1,000** packets we measure ∼59.3 s (Baseline), ∼51.8 s

(Enhanced), and ∼50.2 s (27 states), i.e., about **15**% faster than Baseline for the 27 states model. At **10**,**000** packets the gap widens: ∼275.0 s vs ∼216.9 s vs ∼203.5 s, with the 27 states agent **26**% faster than Baseline and **6**% faster than Enhanced. At **50**,**000** packets the trend persists (∼309.6 s vs ∼246.3 s vs ∼234.9 s), corresponding to **24**% and **4.6**% improvements over Baseline and Enhanced, respectively. In short, reductions in miss chains translate directly into lower processing latency. The *update time* measured on the learning side is essentially flat with respect to the **update interval** (Figure 6.9): each agent shows small fluctuations only, attributable to stochastic action choices.

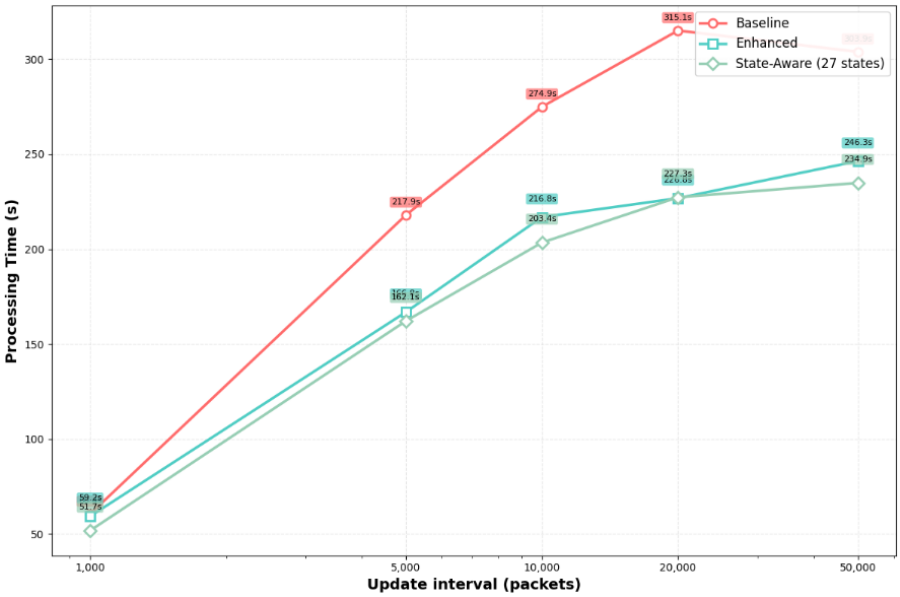


Figure 6.8. Comparation of processing time of RL agents in a digital twin configuration

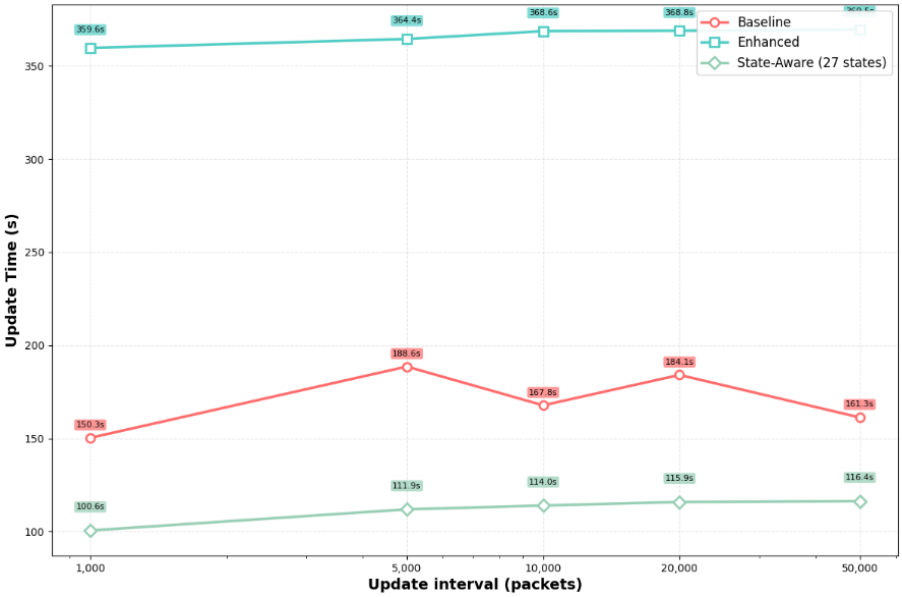


Figure 6.9. Comparation of update time misses of RL agents in a digital twin configuration

The **State-Aware (27 states)** agent exhibits the lowest overhead, the **Baseline** is intermediate, and the **Enhanced** agent is the most expensive. For reference, at **1,000** packets we observe ∼100.6 s (27 states) vs ∼150.3 s (Baseline) vs ∼359.6 s (Enhanced), i.e., the 27 states agent is about **33**% faster than Baseline and **72**% faster than Enhanced. At **50,000** packets the pattern remains (∼116.4 s vs ∼161.3 s vs ∼360.6 s), corresponding to improvements of **28**% (vs Baseline) and **68**% (vs Enhanced). This behavior is consistent with the algorithmic load: the 27 states agent typically requires fewer attempts per packet (fewer Q-updates), whereas the Enhanced variant adds per-attempt computations (adaptive ε, adaptive α, reward shaping), which increase its update overhead despite being stateless.

Across all update intervals, the **27 states** agent delivers the best performance in the digital twin setting: fewer misses, lower processing time, and the smallest agent-side update overhead, consistently outperforming the stateless baselines. Moreover, the **1,000 packets** update interval yields the most favorable timings, as more frequent reorderings limit table staleness and reduce both misses and end-to-end latency (critical path).

Scaling of the best agent (27 states)

We now focus on the **27 states** agent and study how it scales as the rule set grows from **200** to **500** entries, keeping the same update intervals (from 1000 to 50000 packets). For a fixed interval, moving from 200 to 500 rules substantially increases the total number of misses (Figure 6.10 and 6.11). At **1,000** packets, misses rise from ∼3.79 M to ∼25.36 M (**+568%**); at **50,000** packets, from ∼15.74 M to ∼66.48 M (**+322%**). Within each rule size, misses grow monotonically with the interval (e.g., 200 rules: 3.79 M → 15.74 M; 500 rules: 25.36 M → 66.48 M), confirming that longer refresh windows leave the rule table stale for longer.

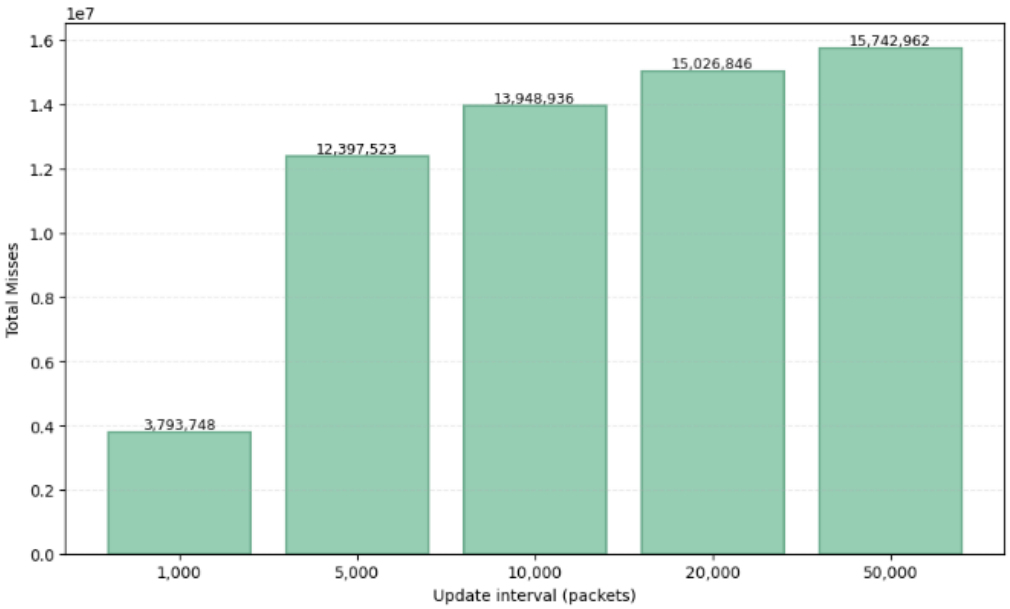


Figure 6.10. 27 states Digital Twin: Total Misses with 200 rules

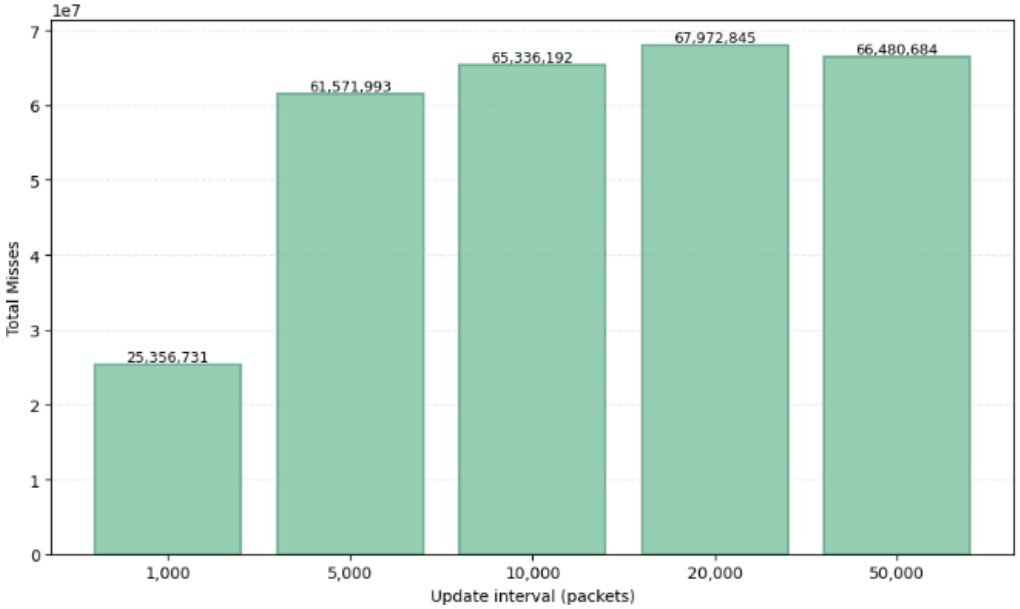


Figure 6.11. 27 states Digital Twin: Total Misses with 500 rules

The processing time of the traditional firewall scales with both the interval and the number of rules. Going from 200 to 500 rules (Figure 6.12 and 6.13), the time at **1,000** packets increases from ∼93.6 s to ∼357.2 s (**+281.6%**), and at **50,000** from ∼308.6 s to ∼914.2 s (**+196.2%**). For a fixed rule size, larger intervals yield longer chains before the first match and thus higher wall-clock time (e.g., 200 rules: 93.6 s → 308.6 s; 500 rules: 357.2 s → 914.2 s).

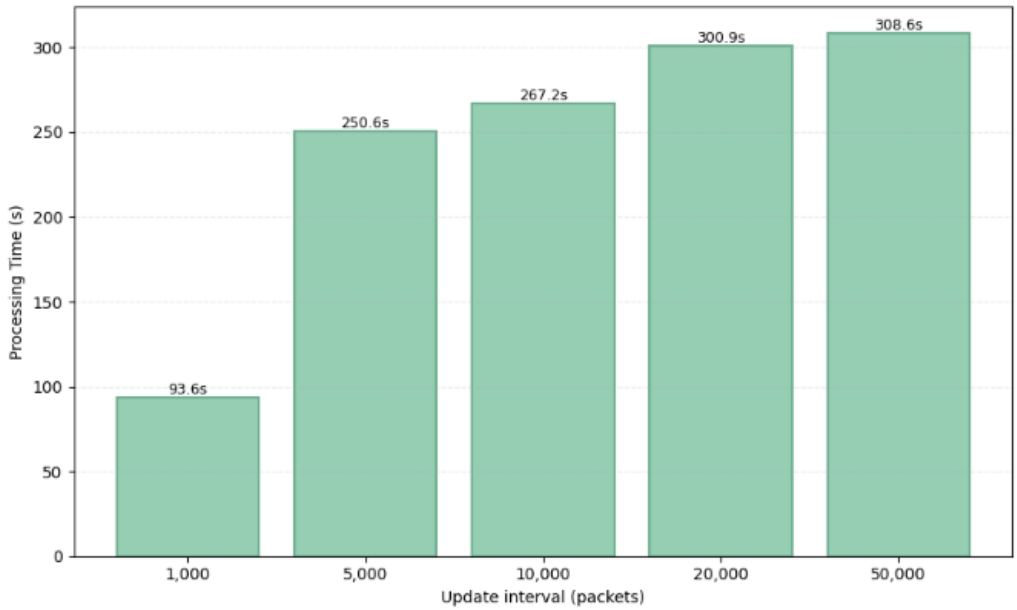


Figure 6.12. 27 states Digital Twin: Processing Time with 200 rules

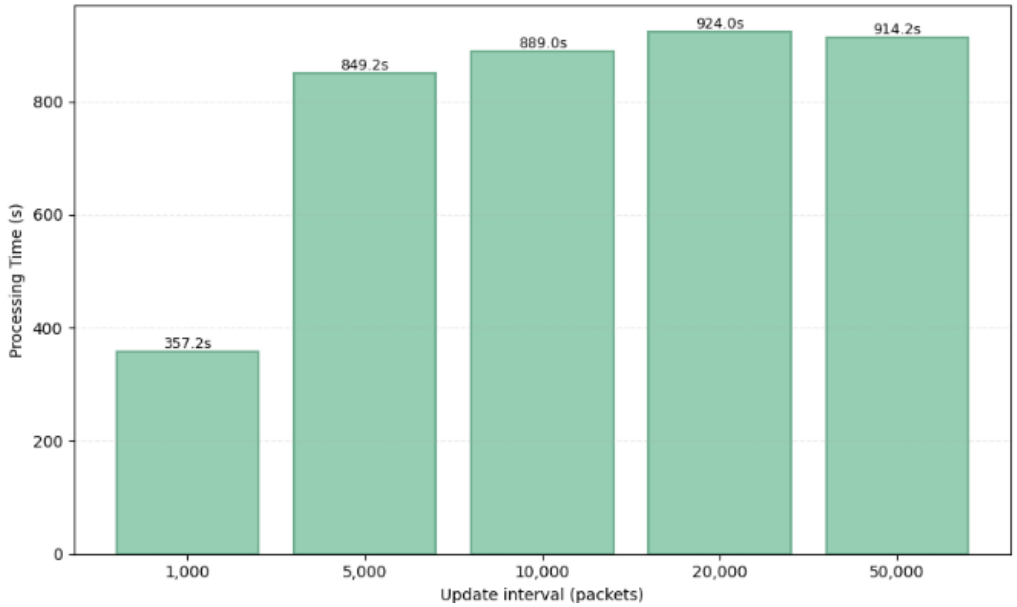


Figure 6.13. 27 states Digital Twin: Processing Time with 500 rules

The RL agent update time is *stable across intervals* for a fixed rule size (Figure 6.14 and 6.15), but increases with the number of rules. It moves from ∼124–137 s (200 rules) to ∼358–387 s (500 rules), i.e., roughly from **+170%** to **+190%**. Consequently, at **200** rules the critical path is agent bound at small intervals (e.g., 1k: 124.6 s vs 93.6 s) and processing-bound at large intervals (e.g., 50k: 133.3 s vs 308.6 s). At **500** rules the two times are comparable at 1k (359.5 s vs 357.2 s), while *processing* dominates beyond 5k (e.g., 50k: 360.5 s vs 914.2 s).

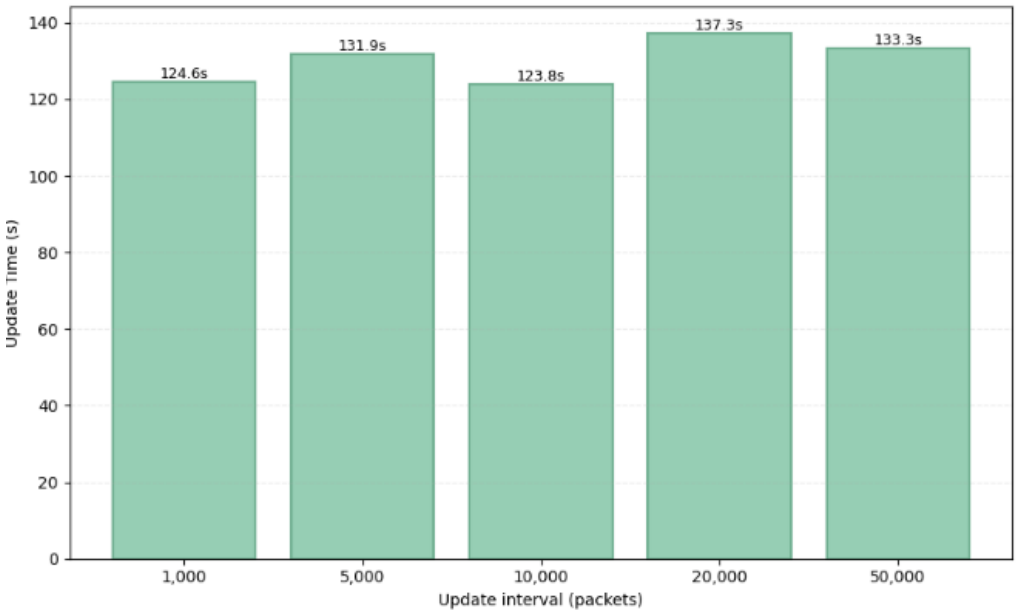


Figure 6.14. 27 states Digital Twin: Update Time with 200 rules

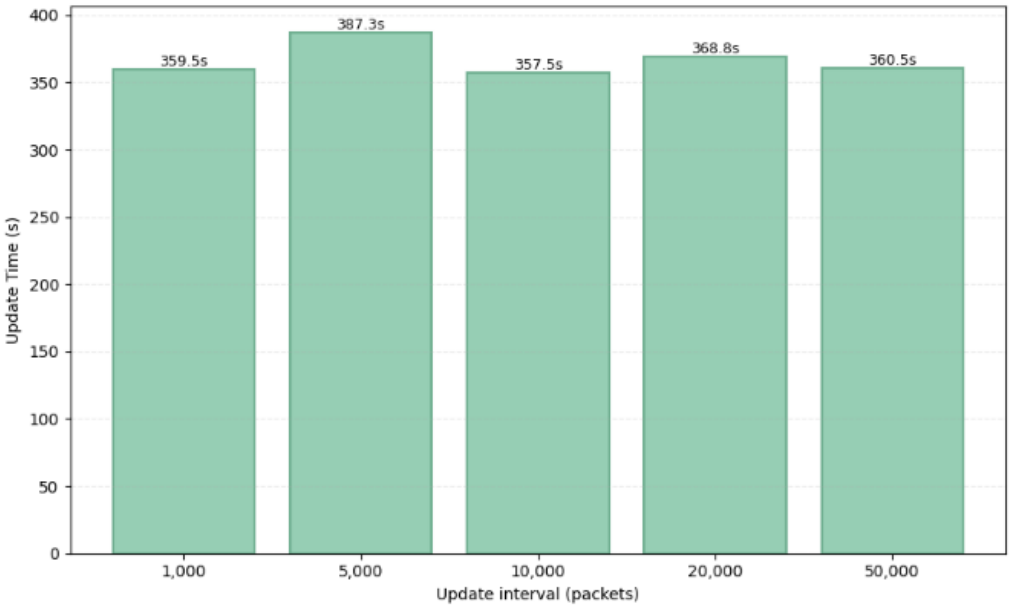


Figure 6.15. 27 states Digital Twin: Update Time with 500 rules

The 27 states agent scales predictably: misses and traditional processing time grow with both rule set size and update interval; agent update time is largely interval agnostic but increases with the rule space. The effective wall-clock per window is governed by the slower of the two components, shifting from agent bound (small intervals, smaller rule sets) to processing bound (large intervals and/or larger rule sets).

6.6.3 Head-to-Head: 27 states Smart Firewall vs 27 states Digital Twin

We compare the best single agent (27 states) against its digital twin counterpart at a fixed update interval of **1,000** packets. For each rule set size we report: the **total misses** and the **total time**. In the digital twin case, the effective wall-clock per window is governed by the *critical path*, i.e.,

$$T_{\mathrm{DT}} \;=\; \max\{\, T_{\mathrm{processing}} \,,\, T_{\mathrm{agent\ update}} \,\}.$$

With 100 rules the single 27 states firewall achieves **486,965** misses and **149.13 s** total time (Figure 6.16 and 6.17). The digital twin records **1,409,220** misses and a critical path time of max(47.74, 75.19) = **75.19 s**. Hence, the single agent is *more accurate*, while the digital twin is *faster* in latency.

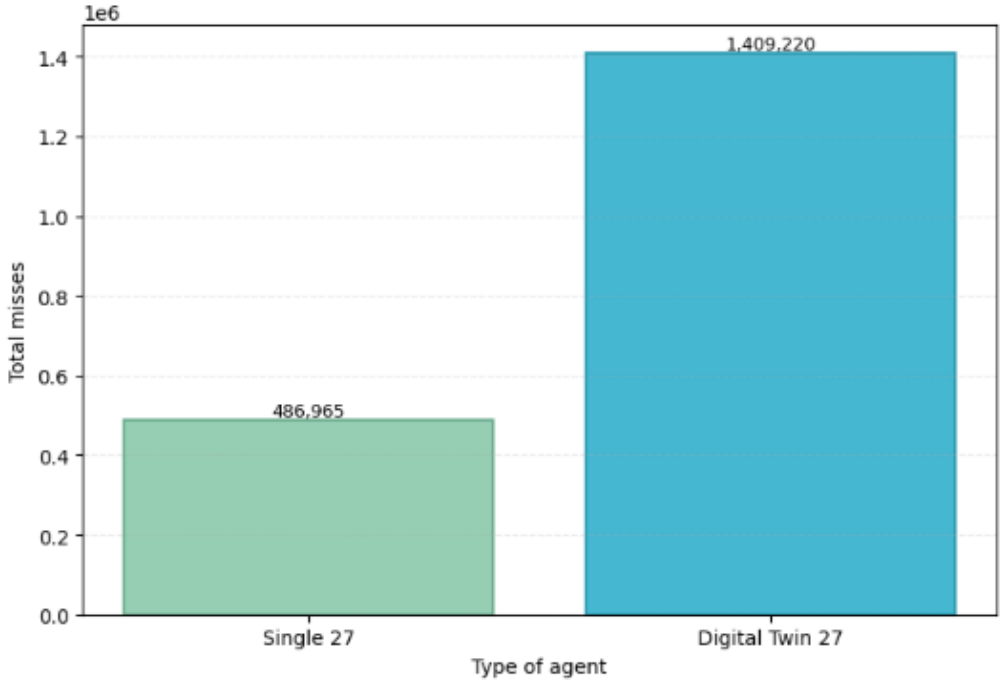


Figure 6.16. Smart Firewall vs. Digital Twin: Total Misses with 100 rules

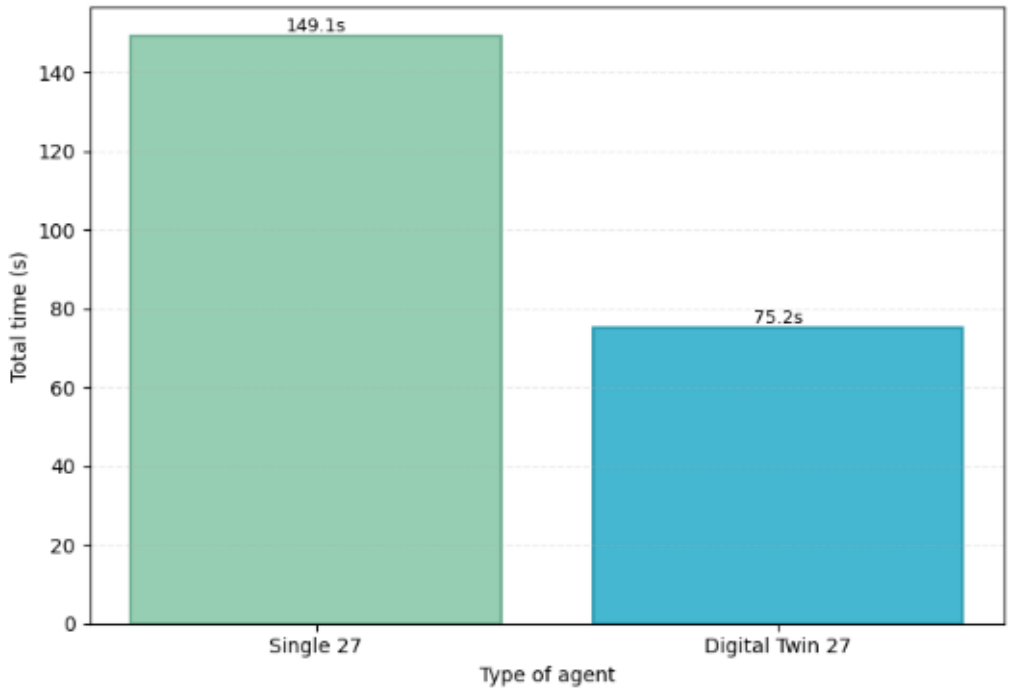


Figure 6.17. Smart Firewall vs. Digital Twin: Total Time with 100 rules

Instead, with 500 rules, the single 27 states firewall attains **3,815,158** misses and **1,157.93 s** total time (Figure 6.18 and 6.19). The digital twin reaches **25,356,731** misses and a critical path time of max(357.21, 359.52) = **359.52 s**. Again, the single agent minimizes misses, whereas the digital twin minimizes the elapsed time.

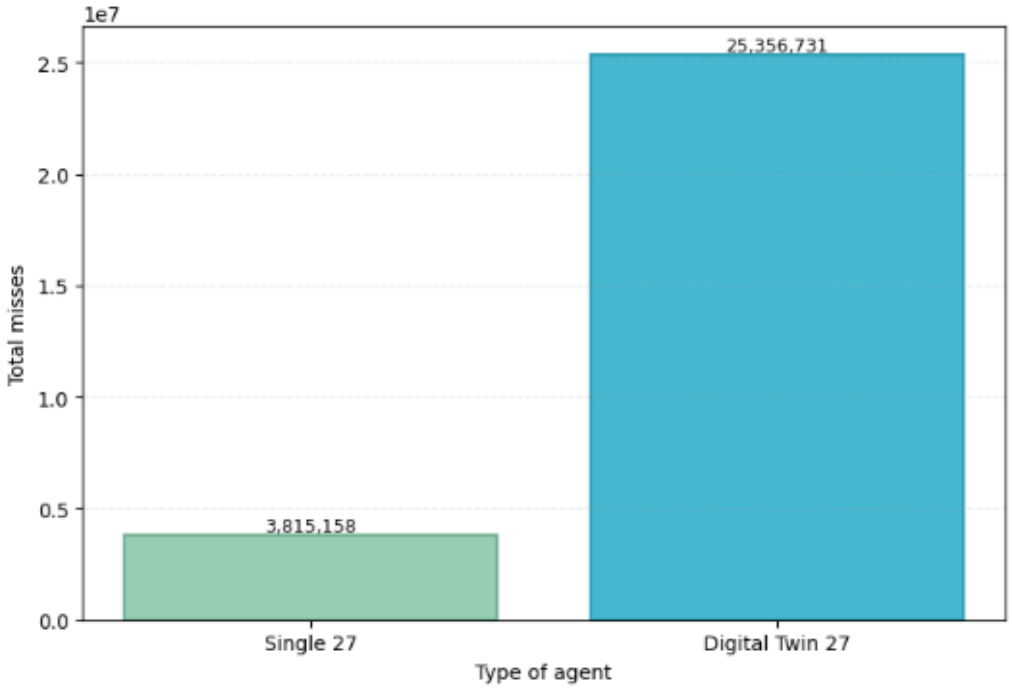


Figure 6.18. Smart Firewall vs. Digital Twin: Total Misses with 500 rules

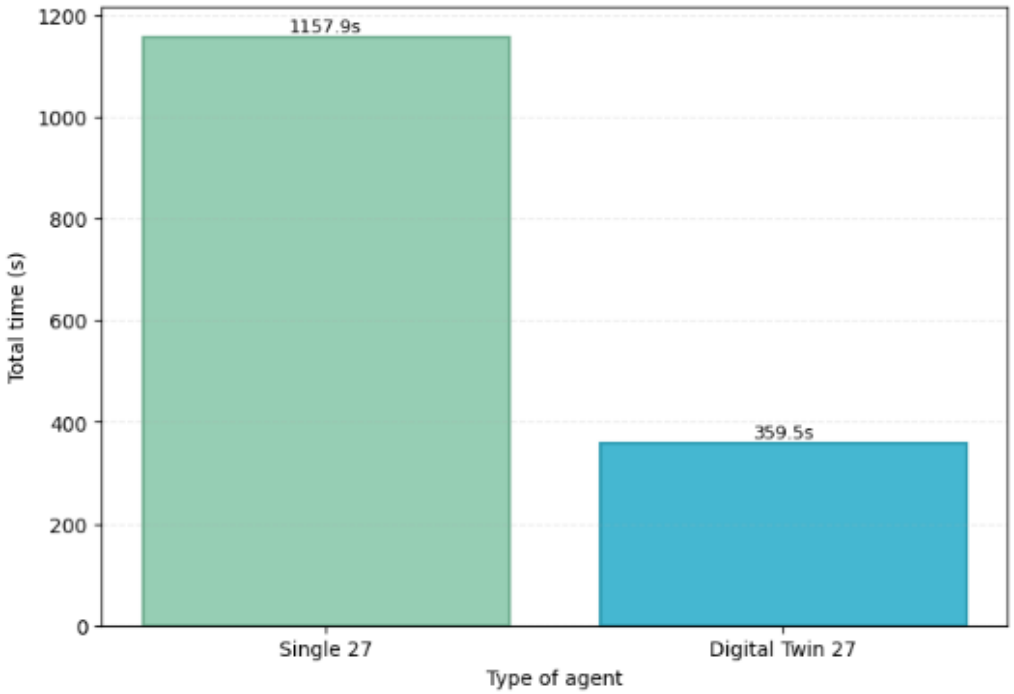


Figure 6.19. Smart Firewall vs. Digital Twin: Total Time with 500 rules

In conclusion, the single 27 states firewall yields the lowest number of misses, while the digital twin offers lower end-to-end time by leveraging parallel processing and periodic rule table refreshes. The preferred solution depends on the target objective: accuracy vs. latency/throughput.

Chapter 7

Conclusions

This thesis examined the role of machine learning in network traffic filtering, arguing that the integration of AI into firewall rule management can materially improve performance under first–match semantics. By framing rule ordering as a reinforcement learning problem, we showed that an agent can internalize traffic regularities and continuously refine the rule sequence to shorten match chains. In practical terms, this translates into reduced latency and greater resilience to bursts, precisely the conditions under which conventional, static orderings tend to create bottlenecks and queue build-ups. The study also explored a digital twin composition in which a learning component evolves the rule order while a traditional firewall enforces decisions, highlighting how learning and enforcement can be decoupled without altering existing operational models.

The overall picture that emerges is that modest amounts of state, coupled with on-line learning, make firewalls more responsive to the temporal structure of real traffic. Rather than relying solely on periodic, manual curation of large rule sets, an RL assisted controller can adapt the ordering to the current mix of protocols, ports, and communication patterns, thus mitigating the cumulative cost of sequential matching. This perspective does not replace traditional policy engineering; instead, it augments it with a mechanism that continuously improves the operational efficiency of the rule base, while remaining compatible with established tooling and first–match policies.

Looking ahead, several extensions appear both natural and valuable. A first direction is *dynamic state abstraction*: instead of fixing the state space a priori, the agent could discover and adapt its granularity to the prevailing traffic, learning when to refine or merge states as conditions evolve. A second avenue concerns *drift–aware operation*, in which exploration, learning rates, and update cadence are adjusted on the fly in response to non-stationarity, preserving stability while remaining agile. A third line is *scale and realism*: validating the approach on substantially larger traces and rule bases, representative of enterprise environments, to stress the learning loop under heavier workloads and more diverse behaviors. Finally, *distributed deployment* offers an architectural pathway to production readiness: coordinating multiple enforcement points and a shared controller, supporting safe, low disruption table swaps, and integrating with existing observability and incident response pipelines. Taken together, these directions suggest that AI assisted

firewalls are not a speculative convenience but a pragmatic step toward networks that adapt their defensive posture as quickly as the traffic that traverses them.

Bibliography

[1] D. Bringhenti, G. Marchetto, R. Sisto, and F. Valenza, "Automation for network security configuration: State of the art and research trends," *ACM Comput. Surv.*, vol. 56, no. 3, Oct. 2023. [Online]. Available: https://doi.org/10.1145/3616401

[2] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," *IEEE Transactions on Dependable and Secure Computing*, vol. 22, no. 3, pp. 2308–2325, 2025.

[3] D. Bringhenti, L. Seno, and F. Valenza, "An optimized approach for assisted firewall anomaly resolution," *IEEE Access*, vol. 11, pp. 119 693–119 710, 2023.

[4] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, and J. Yusupov, "Automated firewall configuration in virtual networks," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 2, pp. 1559–1576, 2023.

[5] D. Bringhenti and F. Valenza, "Greenshield: Optimizing firewall configuration for sustainable networks," *IEEE Transactions on Network and Service Management*, vol. 21, no. 6, pp. 6909–6923, 2024.

[6] ——, "Optimizing distributed firewall reconfiguration transients," *Comput. Netw.*, vol. 215, no. C, Oct. 2022. [Online]. Available: https://doi.org/10.1016/j.comnet.2022.109183

[7] F. Pizzato, D. Bringhenti, R. Sisto, and F. Valenza, "Automatic and optimized firewall reconfiguration," in *NOMS 2024-2024 IEEE Network Operations and Management Symposium*, 2024, pp. 1–9.

[8] D. Bringhenti, F. Pizzato, R. Sisto, and F. Valenza, "Autonomous attack mitigation through firewall reconfiguration," *Int. J. Netw. Manag.*, vol. 35, no. 1, Dec. 2024. [Online]. Available: https://doi.org/10.1002/nem.2307

[9] H. Hamed and E. Al-Shaer, "Dynamic rule-ordering optimization for high-speed firewall filtering," in *Proceedings of the 2006 ACM Symposium on Information, Computer and Communications Security*, ser. ASIACCS '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 332–342. [Online]. Available: https://doi.org/10.1145/1128817.1128867

[10] H. Abie, "An overview of firewall technologies," 12 2000.

[11] D. Bringhenti, S. Bussa, R. Sisto, and F. Valenza, "Atomizing firewall policies for anomaly analysis and resolution," *IEEE Trans. Dependable Secur. Comput.*, vol. 22, no. 3, p. 2308–2325, Nov. 2024. [Online]. Available: https://doi.org/10.1109/TDSC.2024.3495230

[12] F. Valenza and M. Cheminod, "An optimized firewall anomaly resolution," *J. Internet Serv. Inf. Secur.*, vol. 10, pp. 22–37, 2020. [Online]. Available: https://api.semanticscholar.org/CorpusID:212666334

[13] S.-d. Krit and E. Haimoud, "Overview of firewalls: Types and policies: Managing windows embedded firewall programmatically," in *2017 International Conference on Engineering MIS (ICEMIS)*, 2017, pp. 1–7.

[14] R. Hunt, "Internet/intranet firewall security—policy, architecture and transaction services," *Computer Communications*, vol. 21, no. 13, pp. 1107–1123, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S014036649800173X

[15] M. I. Khan, A. Arif, and A. R. A. Khan, "The most recent advances and uses of ai in cybersecurity," *BULLET: Jurnal Multidisiplin Ilmu*, vol. 3, no. 4, pp. 566–578, 2024.

[16] M. F. Ansari, B. Dash, P. Sharma, and N. Yathiraju, "The impact and limitations of artificial intelligence in cybersecurity: a literature review," *International Journal of Advanced Research in Computer and Communication Engineering*, 2022.

[17] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[18] Hugging Face, "Deep reinforcement learning course - unit 1: Rl framework," https://huggingface.co/learn/deep-rl-course/unit1/rl-framework, 2023, accessed: 2025-08-09.

[19] A. D. Kara and S. Yüksel, "Convergence and near optimality of q-learning with finite memory for partially observed models," in *2021 60th IEEE Conference on Decision and Control (CDC)*, 2021, pp. 1603–1608.

[20] C. Hunt, *TCP/IP network administration*. " O'Reilly Media, Inc.", 2002, vol. 2.

[21] S. Naeem, A. Ali, S. Anam, and M. M. Ahmed, "An unsupervised machine learning algorithms: Comprehensive review," *International Journal of Computing and Digital Systems*, 2023.

[22] H. V. Singh, A. Girdhar, and S. Dahiya, "A literature survey based on dbscan algorithms," in *2022 6th International Conference on Intelligent Computing and Control Systems (ICICCS)*, 2022, pp. 751–758.

[23] F. Murtagh and P. Contreras, "Algorithms for hierarchical clustering: an overview," *Wiley interdisciplinary reviews: data mining and knowledge discovery*, vol. 2, no. 1, pp. 86–97, 2012.

[24] IBM. (2023) What is hierarchical clustering? Accessed: 2025-08-24. [Online]. Available: https://www.ibm.com/think/topics/hierarchical-clustering