POLITECNICO DI TORINO
Repository ISTITUZIONALE

TwinWall: A digital twin-driven firewall architecture for adaptive rule ordering optimization

Original
TwinWall: A digital twin-driven firewall architecture for adaptive rule ordering optimization / Bringhenti, D., Valenza, F.. - In: COMPUTER NETWORKS. - ISSN 1389-1286. - ELETTRONICO. - 292:(2027), pp. 1-16. [10.1016/j.comnet.2026.112911]

Availability:
This version is available at: 11583/3017692 since: 2026-11-13T07:28:01Z

Publisher:
Elsevier

Published
DOI:10.1016/j.comnet.2026.112911

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

26 November 2026

Computer Networks 292 (2027) 112911

Contents lists available at ScienceDirect

Computer Networks

journal homepage: www.elsevier.com/locate/comnet

TwinWall: A digital twin-driven firewall architecture for adaptive rule ordering optimization

Daniele Bringhenti *, Fulvio Valenza

Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy

ARTICLE INFO

Keywords:
Firewall
Network security
Digital twin

ABSTRACT

Optimizing the ordering of packet-filtering firewalls is crucial to ensure high packet classification efficiency. In this perspective, the literature has investigated the problem either in a static or batch fashion, by estimating traffic statistics and then computing a new rule ordering offline or on demand, or through dynamic approaches following an estimate-and-optimize paradigm and relying on ad-hoc heuristics based on hand-crafted metrics and tuning choices. Therefore, there is a need for novel algorithms that can learn adaptive rule prioritization strategies directly from experience, possibly in a separate environment from the one responsible for decision making. To this aim, this paper presents TwinWall, a digital-twin-based firewall architecture in which a digital optimizer uses reinforcement learning to derive adaptive rule orderings based on the observed traffic, and periodically exports them to the real firewall. In this way, TwinWall reframes the rule-ordering problem as a continuous learning process, where prioritization emerges directly from traffic interaction rather than from predefined optimization algorithms. A prototype implementation of TwinWall was developed and validated on three datasets with different traffic characteristics, including two real packet traces and one synthetic workload generated with the ClassBench benchmarking tools. The obtained results show that the proposed approach can significantly reduce both the number of rule checks and the corresponding packet processing time, and that it outperforms static and frequency-based baseline strategies.

1. Introduction

Packet-filtering firewalls are network security functions responsible for providing an effective distinction between communications to be permitted for guaranteeing the operation of crucial services and traffic flows to be selectively blocked as potential vectors of malicious activities. This behavior is governed by the definition of a firewall filtering policy, consisting of a set of filtering rules that decide the action, between allow or deny, to be applied to each analyzed packet based on the information contained in its IP 5-tuple header fields, i.e., source and destination IP addresses, source and destination ports, transport protocol. This firewall type is also the most commonly used to protect computer networks, as it is natively available in operating systems and in commercial routers.

However, configuring a packet-filtering firewall policy in such a way as to behave correctly and efficiently has always been a complex task [1]. On the one hand, its definition requires technical knowledge related to IP addresses and ports associated with the traffic, in contrast to web-application firewalls, where more human-intelligible information, i.e., domains and URLs, can be directly specified. On the other hand, most packet filters adopt a first-match resolution strategy, according to which the rules of their policy are analyzed according to their sequential order whenever a packet is processed. The order of rules thus plays a crucial role, because packet processing stops as soon as the first matching rule is found. Therefore, even when the policy is semantically correct, an unfavorable ordering may increase the average number of rule comparisons required to classify packets, with a direct impact on processing latency and overall firewall performance.

In the literature, the first problem has been extensively addressed with success in multiple ways. Conflict analysis techniques such as [2–5] can remove errors present in a manually written firewall policy, so that rule ordering no longer affects the correctness of the policy. Refinement methods such as [6–8] can achieve the same result, by creating the firewall rule set automatically from user-specified intents and providing formal guarantees of its conflict-freedom. Conversely, the second problem remains only partially addressed. In particular, the most relevant open challenge is to understand how firewall rules can be reordered dynamically according to the traffic currently observed, while maintaining high performance under non-stationary traffic conditions. Existing dynamic approaches typically follow an estimate-and-optimize paradigm: they first extract traffic statistics, such as rule

* Corresponding author.
E-mail addresses: daniele.bringhenti@polito.it (D. Bringhenti), fulvio.valenza@polito.it (F. Valenza).

https://doi.org/10.1016/j.comnet.2026.112911
Received 30 April 2026; Received in revised form 13 October 2026; Accepted 2 November 2026
Available online 9 November 2026
1389-1286/© 2026 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

matching frequencies or recency indicators, and then periodically recompute the rule order on the basis of these measurements. As a result, they rely on ad-hoc heuristics that heavily depend on handcrafted metrics and tuning choices. This limits their flexibility in rapidly changing scenarios, where the relationship between observed traffic and an effective rule ordering may be too complex to be captured by manually designed optimization logic alone.

Hence, there is a need for approaches that can move beyond statistics-driven reordering and instead learn adaptive rule prioritization strategies directly from experience. Rather than separating traffic observation from optimization, these approaches should integrate both aspects into a single decision-making process, capable of continuously refining reordering choices over time. In this way, adaptation could emerge from interaction with the traffic itself, instead of being constrained by predefined metrics, thresholds, or update guidelines. This requirement is particularly relevant in operational environments where traffic conditions may change significantly over time, making it difficult to capture their dynamics through fixed indicators alone. A learning-based approach could therefore provide a more flexible and effective way to adjust firewall rule ordering to the actual traffic being processed.

In order to fill this gap in the literature, this paper proposes TwinWall, a digital-twin-driven architecture for adaptive firewall rule reordering. Specifically, the TwinWall design is composed of two cooperating modules, both observing the same traffic: a standard sequential packet-filtering firewall and its virtual digital twin. The former is the component responsible for enforcing filtering decisions on the real packets, whereas the latter acts as an intelligent replica that observes the traffic and uses reinforcement learning to derive an adaptive rule reordering strategy, with actions evaluated through rewards reflecting their impact on packet classification efficiency. At regular intervals, the rules reordered by the digital twin are transferred to the real firewall, which starts to use them to filter the next received packets.

The novelty of this design is twofold. First, TwinWall introduces a reinforcement-learning formulation of the firewall rule-ordering problem in which rule prioritization is learned directly through packet-level interaction. The novelty derived from using this technique lies in the mapping of firewall classification to a sequential decision process: the traffic abstraction defines the learning state, the selection of the next rule represents the action, and the reward directly reflects the number of unsuccessful rule evaluations required before classification. Consequently, the deployed ordering is not computed by applying a predefined optimization heuristic to previously aggregated traffic statistics, but it emerges from the action values progressively learned from the observed packets. This formulation also enables the same learning mechanism to derive either a single global ordering or multiple traffic-state-conditioned orderings, without requiring a different optimization algorithm for each traffic class. Second, the digital twin decouples this reinforcement-learning process from enforcement. While the real firewall continues to operate with the currently deployed rule set, the twin can safely explore alternative rule priorities and refine the learned action values in parallel, without introducing exploration or reward computation into the live packet-processing path. The resulting rule orderings are then periodically transferred to the real firewall through a policy-view replacement. Therefore, TwinWall combines a novel learning-based formulation of adaptive rule ordering with an architectural separation that makes reinforcement learning compatible with the latency and predictability requirements of conventional linear firewalls.

A prototype implementation of the proposed firewall architecture was validated on three datasets with different traffic characteristics. Two datasets consist of real packet traces collected in heterogeneous backbone and industrial-control environments. Since these traces do not include corresponding firewall policies, a traffic-driven rule generator was developed to automatically derive a coherent rule set from each dataset, thus ensuring consistency between the observed packets and the generated policy. The third dataset is instead a synthetic benchmark workload generated using the ClassBench toolchain. Multiple configuration settings for the digital twin were then tested in order to assess the most efficient one. The best-performing setup was subsequently compared with non-reinforcement-learning baseline strategies, e.g., frequency-based approaches, and showed superior performance in terms of misses and packet classification time. Finally, a comparative analysis with related approaches was carried out to discuss differences in computational workflow, deployment requirements, and performance.

The remainder of this paper is structured as follows. Section 2 discusses related work on the firewall rule reordering problem. Section 3 presents the design of TwinWall. Section 4 formalizes the reinforcement learning algorithm used in the digital twin. Section 5 describes the implementation of a prototype featuring the proposed design, and reports the results of the conducted validation campaign. Section 6 concludes the paper and discusses future work.

2. Related work

The firewall rule reordering problem was originally formulated as the Optimal Rule Ordering (ORO) problem in [9], with the objective of finding a legitimate ordering of a first-match rule set that minimizes the average packet-classification cost.

A first group of studies in the related literature addresses this problem in a static or batch way: traffic statistics are first estimated, then a new ordering is computed offline or on demand and deployed afterwards. Among early studies, [10] models rule precedence relationships through a directed acyclic graph and uses a simple sorting heuristic that exchanges adjacent rules according to their matching probabilities. In [11], the ORO problem is formulated as a constrained combinatorial optimization problem and solved through simulated annealing. Starting from a feasible topological order, the ROSA optimizer explores neighboring precedence-preserving permutations through rule swaps, seeking to reduce the expected search depth. [12] proposes the SubGraph Merging (SGM) heuristic, which evaluates groups of mutually constrained rules through their average matching probabilities and incrementally constructs a precedence-preserving ordering. Instead, a dependency-aware heuristic approximation algorithm is proposed in [13], so as to minimize a trace-weighted matching-cost objective by using the observed rule frequencies.

Subsequently, this batch-optimization research line for solving the ORO problem was further improved with more specialized and efficient resolution algorithms. In particular, the graph-based heuristics proposed in [14,15] are able to operate on rule-dependency structures more efficiently, by replacing adjacency matrices with adjacency lists, and by estimating rule importance by also considering directly dependent rules. Next, a set of studies [16–18] move beyond the original strict ORO and introduce the Relaxed ORO (RORO) problem, where dependent rules may still be reordered provided that policy equivalence is preserved. To make this feasible, the approaches introduced in those papers use Zero-suppressed Decision Diagrams (ZDDs) both to recompute post-reordering packet-evaluation sets and to test whether local moves preserve semantics, combining these checks with heuristic moves, pairing strategies, and simulated annealing. More recently, metaheuristic solutions [19,20] were investigated for the ORO problem. The initial approach combines a genetic algorithm with dedicated mechanisms for preserving rule-precedence constraints, whereas its extended version adopts a two-stage procedure: a DAG-based topological ordering first prioritizes constrained rules according to their activation frequencies, and a genetic algorithm subsequently combines this constrained ordering with the unconstrained rules to optimize the complete rule list. Although these methods provide effective solutions to the static rule-ordering problem, they operate as one-shot batch optimizers and require rule-activation frequencies as input to the optimization process. A different approach is pursued by FROG [21],

which exploits rule similarity to construct an ordering organized into blocks that can be skipped by firewall scanners such as OpenBSD Packet Filter (PF) [22]. Although FROG does not require rule-activation frequencies, it remains an offline optimizer and its performance benefits rely on a skip-aware scanning engine, limiting its direct applicability to conventional sequential first-match firewalls.

A second group of studies addresses firewall rule reordering in a dynamic or periodic way, i.e., by continuously adapting the rule order over time rather than computing it only once offline. In [9], a traffic-aware mechanism is introduced to dynamically update rule priorities on the basis of online estimates of rule matching frequency and recency. Reordering is then triggered either periodically or when a degradation in filtering performance is observed. Instead, a window-based method is introduced in [23], using traffic histograms collected during previous windows to dynamically reorder not only firewall rules, but also the fields within the rules. Finally, the swapping-window techniques described in [24,25] are specifically designed for dynamic network environments, where the rule set is progressively reordered either after a fixed number of packet matches or in response to observed performance trends, while traffic statistics are maintained through lightweight estimators to limit monitoring overhead. Overall, these approaches show that adaptive reordering can substantially reduce the average packet-classification cost when traffic characteristics evolve over time. However, similarly to the static methods discussed above, they still follow an estimate-and-optimize paradigm: traffic behavior is first summarized through manually designed indicators, such as frequencies, recency measures, or histogram statistics, and only afterwards a separate optimization step is executed to compute the next rule ordering.

A partially different direction is represented by methods that improve packet classification without directly solving the rule-reordering problem. For example, the data-mining-based oracle proposed in [26] tries to predict the most likely matching rule and falls back to sequential inspection if the prediction is wrong. While effective, this approach leaves the original rule ordering unchanged and accelerates filtering through an auxiliary classifier rather than through adaptive reordering of the policy itself.

Compared with the above literature, TwinWall occupies a different point in the design space. Like prior dynamic approaches, it targets traffic-dependent adaptation rather than one-shot optimization; however, it does not periodically recompute an ordering from manually designed statistics. Instead, it uses reinforcement learning to learn rule-prioritization policies directly from experience: TwinWall formulates rule ordering as a packet-level sequential decision process in which rule selections are the actions and classification depth directly shapes the reward, so that the ordering is extracted from learned action values rather than computed from an explicit traffic-frequency objective. Moreover, learning is decoupled from enforcement through a digital twin: the optimizer refines candidate orderings in a shadow environment, while the production firewall continues to process live traffic with a standard first-match engine until a new ordering is atomically deployed. Therefore, TwinWall differs from prior static approaches because it is inherently adaptive, and from prior dynamic approaches because it replaces estimate-and-optimize logic with direct learning inside a digital-twin architecture.

3. The proposed architecture design

TwinWall is designed as a two-module architecture in which traffic-driven learning is decoupled from the enforcement of filtering decisions. As shown in Fig. 1, the system consists of a *Digital Twin Optimizer* (DTO), which observes the traffic and learns how firewall rules should be prioritized, and a *Rule Execution Engine* (REE), which makes filtering decisions on the live traffic. The digital twin is not merely an execution environment surrounding the learning algorithm, but the architectural element that enables adaptive learning without altering

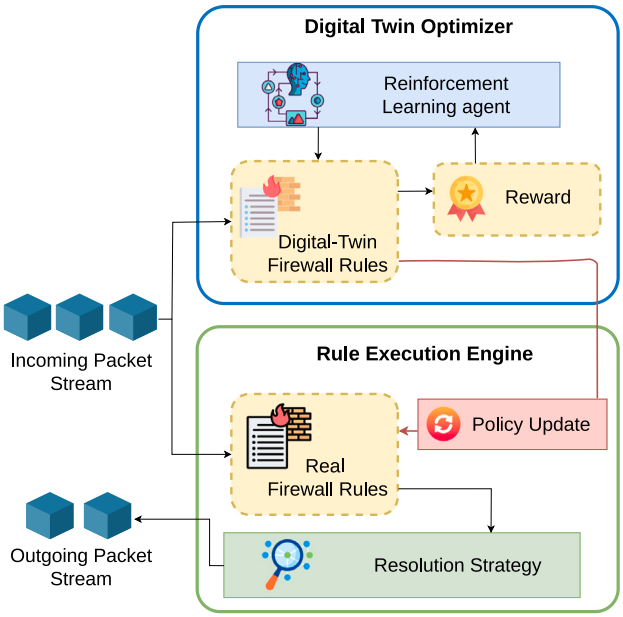


Fig. 1. The TwinWall design.

the operational behavior of the production firewall. Reinforcement learning requires exploration, evaluation of potentially suboptimal actions, reward computation, and repeated updates of the learned values. Performing these operations directly in the enforcement path would make packet-processing latency dependent on the learning procedure and could expose live traffic to exploratory rule selections. This separation ensures that exploratory rule selections remain confined to the DTO and never affect live filtering decisions. The REE receives only complete rule orderings at predefined update points and continues to process packets according to deterministic first-match semantics.

3.1. Digital Twin Optimizer

The DTO is the learning component of TwinWall. Its purpose is not to enforce filtering decisions on production packets, but to observe the same traffic processed by the REE and learn which rules should be assigned higher priority so as to reduce the average packet classification cost. To achieve this objective, the DTO adopts a tabular Q-learning strategy, i.e., a reinforcement learning method in which an agent gradually estimates the utility of taking different actions in different states on the basis of the rewards collected during interaction with the environment. Concretely, this knowledge is stored in a *Q-table*, whose entries associate each state–action pair with a learned utility value named *Q-value*, and is used to derive a policy, i.e., a strategy for selecting which rule should be tested next in each traffic condition. A key advantage of Q-learning in this context is that it does not require an explicit model of the traffic dynamics: the DTO does not need to predict in advance how traffic will evolve, but can instead learn directly from the packet stream it observes.

In TwinWall, the Q-learning elements are mapped to the firewall rule reordering problem. The *environment* is the packet classification process induced by the observed traffic. The *state*[1] is an abstract representation of the class of the packet currently being processed. The *action*

[1] To avoid ambiguity, it is worth underlining that the notion of *state* used in TwinWall is purely related to the reinforcement learning model and does not refer to stateful firewalling semantics. The firewalls considered in this study remain stateless packet filters, since filtering decisions are taken solely on the basis of packet header fields.

is the choice of the next rule to evaluate among the rules of the filtering policy. The *reward* expresses how beneficial that choice was for packet classification: selecting a rule that matches the packet leads to positive feedback, whereas selecting a non-matching rule produces a penalty, which can be further increased when multiple attempts are required for the same packet. Finally, the *policy* is the rule-selection strategy derived from the learned Q-values. Accordingly, each entry of the Q-table measures how promising it is to evaluate a given rule under a given traffic state. Over time, rules that tend to match early for packets of a certain class accumulate higher values, while rules that frequently cause unsuccessful attempts receive lower ones.

The choice of the state abstraction determines the granularity with which the DTO can distinguish among different classes of traffic, and therefore the number of rows of the Q-table. In general, the number of states can grow according to how finely packet characteristics are partitioned. When a single state is used, all packets are treated as belonging to the same class, and the DTO learns one global rule prioritization strategy for the entire traffic. When multiple states are introduced, instead, the DTO can learn different rule preferences for different packet classes. For example, a first solution may distinguish packets only by transport protocol, thus using separate states for TCP, UDP, and other traffic, which may include ICMP and any remaining protocols. A richer abstraction may further subdivide packets according to the category of the source port, for instance by distinguishing system/well-known, user/registered, and dynamic/private ports. An even more expressive abstraction may also include the destination-port category, so that packets are grouped according to the joint combination of protocol, source-port tier, and destination-port tier.

Operationally, the DTO processes traffic as a continuous learning stream. When a packet is observed, the DTO first maps it to its abstract state and then starts an iterative rule-evaluation process. At each step, it selects one candidate rule using a state-conditioned ϵ-greedy policy: with high probability it exploits the rule that currently has the highest learned value for that state, while with a smaller probability it explores alternative rules in order to avoid premature convergence to suboptimal priorities. If the selected rule matches the packet, the DTO receives a positive reward, the packet is considered classified, and the learning process moves to the next packet. If the rule does not match, the DTO receives a penalty, records that unsuccessful attempt, and selects another rule among those not yet tried for the same packet. In this way, the learning process mirrors the sequential nature of first-match firewalls: long within-packet search chains are discouraged, whereas rules that tend to resolve packets quickly are progressively favored. This procedure also represents the key difference with respect to traditional estimate-and-optimize approaches. In those methods, traffic statistics are first collected and then used by a separate optimizer to compute a new ordering. In TwinWall, instead, observation and optimization are integrated into a single learning loop: each packet contributes directly to refining the rule-selection policy.

Periodically, the DTO converts the learned Q-values into deployable rule orderings for the REE. In the single-state case, this produces a single ordered rule list obtained by sorting the rules according to their learned utility. In the state-aware variants, the DTO instead derives a set of state-conditioned ordered lists, one for each abstract traffic state, by sorting the corresponding row of the Q-table in descending order.

3.2. Rule Execution Engine

The REE is the production-side component of TwinWall. Its function is to enforce the filtering policy on live traffic by preserving the standard operational model of a sequential packet-filtering firewall. More precisely, the REE applies first-match semantics: for each incoming packet, it scans the currently active rule ordering from top to bottom and enforces the action associated with the first matching rule. In this way, TwinWall retains the simplicity, determinism, and compatibility of conventional linear firewalls, without requiring changes to their basic enforcement logic.

Instead of relying on a fixed administrator-defined sequence, the REE receives rule orderings periodically produced by the DTO. At startup, the REE is initialized with a valid rule arrangement, which may either coincide with the original policy order or correspond to a preliminary ordering already learned by the DTO. During operation, the REE continues to process packets using the currently deployed ordering, while the DTO simultaneously observes traffic and refines a new one in parallel. Once a new ordering becomes available, the DTO replaces the active rule view of the REE with the updated one for subsequent packets. This update mechanism is naturally designed as an atomic replacement of the current ordered view, so that learning and enforcement remain decoupled and packet processing is not interrupted during reconfiguration.

In greater detail, the REE may be configured either with a single global rule ordering or with multiple orderings associated with different abstract traffic classes, depending on the output exported by the DTO. In the simplest case, the REE maintains only one ordered rule list, which is applied uniformly to all packets. In the more general configuration, instead, the REE stores a small collection of ordered rule views, one for each traffic class considered by the DTO. Upon packet arrival, the REE first performs a lightweight dispatch operation based on the same packet features used by the DTO to define the traffic classes, and uses the result to select the corresponding rule ordering. This dispatch mechanism can be implemented as a very small lookup structure, with one entry per traffic class, that redirects the packet classification process to the appropriate rule list. Once the corresponding ordering has been selected, packet processing proceeds according to ordinary first-match semantics. Therefore, TwinWall preserves the simplicity and predictability of sequential packet filtering while still enabling traffic-dependent prioritization of rules.

Keeping the REE free from reinforcement-learning logic is a central architectural requirement of TwinWall. If learning were embedded directly in the packet-processing path, each packet could require online action selection, reward handling, Q-value updates, and exploratory rule evaluations before a filtering decision is reached. The runtime behavior of the production firewall would therefore depend on the learning phase and on the current exploration policy. The digital-twin separation prevents this coupling: all learning-related operations are confined to the DTO, while the REE performs only a lightweight state-based dispatch, when required, and ordinary sequential matching over an already exported rule ordering. Moreover, the DTO does not directly impose individual exploratory actions on the REE, but transfers complete rule views only at predefined update points, allowing their deployment through an atomic replacement. Consequently, the digital twin provides the isolation boundary between adaptive optimization and operational enforcement, preserving a stable and predictable processing path while enabling the rule priorities to evolve continuously.

3.3. Deployment considerations

The deployment of TwinWall requires three main aspects to be considered: how the DTO observes the traffic processed by the firewall, where the DTO is executed, and how the rule orderings generated by it are transferred to the REE.

First, the DTO must receive traffic observations that are representative of the packets processed by the REE. Depending on the available infrastructure, this information may be obtained through packet mirroring, traffic replication, or telemetry exported by the firewall platform. Packet mirroring may provide the DTO with packet-level information without placing it in the production forwarding path, whereas exported telemetry may reduce monitoring and communication overhead at the cost of providing a less detailed traffic view. In any case, the DTO and

the REE do not need to process each packet in strict lockstep, since the DTO does not participate in live filtering decisions.

Second, the DTO may be executed either externally or within the firewall platform. An external DTO can run on a dedicated server or network controller and receive mirrored traffic or exported telemetry from the production firewall. This arrangement preserves a strong separation between learning and enforcement and avoids consuming computational resources on the firewall appliance. Although it introduces communication latency, this does not directly affect production packet processing and may only delay the deployment of a newly generated rule ordering. Alternatively, when supported by the platform, the DTO could run on the firewall control plane or within an isolated on-device environment. This arrangement could reduce communication latency and provide more direct access to traffic observations and policy-management functions, but it would require sufficient computational resources. In either case, reinforcement-learning operations remain outside the live enforcement path.

Third, the rule orderings produced by the DTO must be translated into the native policy representation of the target firewall and installed without exposing an incomplete configuration. For software-based firewalls that provide programmable configuration interfaces, a platform-specific adapter can reconstruct the ordered rule chain, generate a replacement configuration, or invoke a management API. Vendor-managed firewall appliances may instead require additional control-plane operations, including translation into a proprietary policy model, policy validation or compilation, staging, and configuration commit. Moreover, whenever supported, the new rule view should be prepared separately and activated through a transactional or atomic replacement, while the firewall continues to process packets using the previously deployed ordering.

4. Algorithm formalization

This section formalizes the reinforcement-learning algorithm adopted by the DTO for establishing the rule orderings to be exported to the REE.

4.1. Packet and firewall rule models

The packet space P is modeled in such a way that each $p \in P$ is a packet class identified by an IP 5-tuple, i.e., $p = (\text{srcIP}, \text{dstIP}, \text{srcPort}, \text{dstPort}, \text{proto})$. Each component of p can only be a single value (e.g., 10.10.2.3 for the source or destination IP address, 80 for the source or destination port, TCP for the protocol).

A firewall rule set R of size $N \in \mathbb{N}$ is instead modeled as $R = \{r_1, r_2, \ldots, r_N\}$. Each rule $r_i \in R$ is a pair $r_i = (c_i, a_i)$. Specifically, c_i is a predicate over packet-header fields, modeled as a 5-tuple similarly to p, with the main difference that each component may also represent a range of values (e.g., 10.10.2.0/24 as IP address fields, [11000, 12000] as port fields, TCP, UDP as protocol). Then, $a_i \in \{\text{allow}, \text{deny}\}$ is the associated action of r_i to be performed to the packets matching the rule condition c_i.

Given these two models, the rule-matching predicate $match\colon P \times R \to \mathbb{B}$ maps an input pair composed of a packet $p \in P$ and a rule $r_i \in R$ to true if p satisfies the condition c_i, i.e., if the specific value carried by p in each 5-tuple field lies within the corresponding (possibly more general) set/range specified by c_i.

At runtime, packet filtering is performed over an ordered view of the rule set. Formally, an ordering is a sequence $\pi = (r^{(1)}, r^{(2)}, \ldots, r^{(N)})$, where each rule of R appears exactly once. Under first-match semantics, the first matching rule for packet p under ordering π is $r^{\star}_{\pi}(p) = r^{(j^{\star})}$, where $j^{\star} = \min\{j \in \{1, \ldots, N\} \mid \text{match}(p, r^{(j)}) = \text{true}\}$. The firewall therefore enforces the action of $r^{\star}_{\pi}(p)$. In the general multi-state setting adopted by TwinWall, multiple orderings may coexist in the REE, one for each abstract traffic state. However, each of them remains a complete ordering of the same common rule set R.

Any reordering of R is semantically admissible under the condition of conflict freedom, i.e., for any $p \in P$, there do not exist two rules $r_i, r_j \in R$, such that $a_i \neq a_j$ and $match(p, r_i) = match(p, r_j) = true$. This condition is not restrictive, as in the literature there are several methodologies that can be applied to remove conflicts in firewall rule sets [2–5].

4.2. Traffic-state abstraction

The reinforcement learning module of TwinWall does not operate directly on raw packets, but on a finite set of abstract traffic states derived from packet header information. Let $\phi\colon P \to S$ be the state abstraction function, mapping each packet class $p \in P$ to a discrete traffic state $s \in S$, where S is a finite state space.

The role of ϕ is to partition the observed traffic into classes that are expected to exhibit different rule-prioritization behaviors. Therefore, the choice of ϕ determines the granularity with which the DTO can distinguish among traffic patterns, and consequently the number of rows of the learned Q-table. Different abstractions can be adopted depending on the desired trade-off between expressiveness and simplicity. In the simplest case, all packets are mapped to the same symbolic state, i.e., $|S| = 1$, and the optimizer learns a single global rule prioritization strategy. A more informative abstraction distinguishes packets according to their transport protocol, thus yielding three states corresponding, for instance, to TCP, UDP, and OTHER traffic. Additional discrimination can be obtained by partitioning source and destination ports into coarse categories such as system/well-known, user/registered, and dynamic/private ports. In this way, the state space can be refined from 3 states to 9 states, and then to 27 states, by progressively combining protocol class, source-port tier, and destination-port tier. The resulting abstraction allows TwinWall to learn different rule priorities for different traffic classes, such as TCP packets directed to well-known service ports or UDP packets using dynamic port ranges.

4.3. Reinforcement learning interaction model

Traffic is modeled as a sequence of packet classes $(p_t)_{t \geq 1}$, where each $p_t \in P$ is processed independently according to first-match firewall semantics. For each packet p_t, the DTO performs an iterative search over the firewall rule set until a matching rule is found. This search is modeled as a reinforcement-learning interaction.

At any step of the interaction, the current reinforcement-learning state is $s_t = \phi(p_t) \in S$, i.e., the abstract traffic class associated with the current packet. The action space coincides with the rule set itself R. Hence, selecting an action means selecting the next firewall rule to be evaluated for the current packet.

Since the same rule should not be tested multiple times on the same packet, TwinWall maintains, for each packet p_t, the set $T_t \subseteq R$ of rules already evaluated unsuccessfully for that packet. The set is initialized as $T_t = \emptyset$ when the processing of p_t begins and is updated after every unsuccessful rule evaluation as

$$T_t \leftarrow T_t \cup \{r_{t,h}\}. \tag{1}$$

At the hth rule-selection step, the set of admissible actions is therefore

$$A_{t,h} = R \setminus T_t. \tag{2}$$

The DTO selects the next rule $r_{t,h}$ exclusively from $A_{t,h}$. If the selected rule matches p_t, the search terminates and the interaction moves to the next packet. Otherwise, the rule is added to T_t according to (1), and the interaction continues on the same packet.

This interaction model can be viewed as a controlled process over an augmented decision context that includes both the traffic state s_t and the set T_t of rules already evaluated unsuccessfully for the current packet. However, TwinWall deliberately learns Q-values only over the reduced state–action space $S \times R$, i.e., the value function is indexed as $Q(s, r)$ and does not include T_t as part of its index. The set T_t is

instead maintained only during the processing of the current packet and is used to construct the admissible action set $A_{t,h}$ according to (2). This design keeps the learned representation compact while ensuring that unsuccessful rule evaluations are not repeated within the same packet-classification process.

4.4. Reward function

The reward function is designed to favor rule selections that lead to early packet classification. Let $\ell_{t,h}$ denote the number of unsuccessful rule proposals already made on packet p_t before the hth rule-selection step. If the rule selected at step h for packet p_t is denoted by $r_{t,h} \in R$, the associated reward is

$$\rho_{t,h} = \begin{cases} r^+ - \lambda \ell_{t,h}, & \text{if match}(p_t, r_{t,h}) = \text{true}, \\ r^- - \lambda \ell_{t,h}, & \text{otherwise}, \end{cases} \tag{3}$$

where $r^+ > 0$ is the positive reward assigned to a successful match, $r^- < 0$ is the penalty associated with an unsuccessful proposal, and $\lambda \geq 0$ controls the additional penalty for long within-packet search chains.

According to Eq. (3), TwinWall does not merely reward correctness, which is already guaranteed by the conflict-free assumption on the firewall rule set, but also rewards efficiency. A rule that matches only after many previous failures is less valuable than a rule that resolves the packet immediately. In this way, the learning objective is naturally aligned with the minimization of packet classification cost.

In the multi-state variants of the proposed reinforcement-learning model, successful rule matches receive a positive base reward, whereas non-matching proposals receive a negative base reward. In both cases, an additional penalty proportional to the number of already failed attempts is applied. This choice explicitly discourages long sequential searches and promotes rule orderings that reduce average search depth.

4.5. Q-learning policy and rule-ordering extraction

For each state $s \in S$ and rule $r \in R$, the DTO learns an action-value function $Q : S \times R \to \mathbb{R}$, where $Q(s, r)$ estimates the utility of evaluating rule r when the current packet belongs to traffic class s. Collecting all such values generates a Q-table with $|S|$ rows and N columns.

Action selection follows an ϵ-greedy policy with within-packet masking. At the hth rule-selection step for packet p_t, the optimizer considers only the admissible action set $A_{t,h}$ defined in (2). With probability ϵ, the DTO explores by selecting one rule uniformly at random from $A_{t,h}$. With probability $1 - \epsilon$, it exploits the current knowledge by selecting an admissible rule with maximum Q-value:

$$r_{t,h} \in \arg\max_{r \in A_{t,h}} Q(s_t, r). \tag{4}$$

The ϵ-greedy mechanism therefore controls exploration and exploitation, while the action mask guarantees that a rule already evaluated unsuccessfully for the current packet cannot be selected again.

After each interaction step, the Q-value associated with the selected state–rule pair is updated according to the standard tabular Q-learning rule

$$Q(s_t, r_{t,h}) \leftarrow Q(s_t, r_{t,h}) + \alpha \Big[\rho_{t,h} + \gamma \max_{r \in R} Q(s', r) - Q(s_t, r_{t,h}) \Big], \tag{5}$$

where $\alpha \in (0, 1]$ is the learning rate and $\gamma \in [0, 1)$ is the discount factor. Here, s' denotes the state observed after evaluating rule $r_{t,h}$. If the selected rule matches the current packet, then $s' = \phi(p_{t+1})$, i.e., the interaction advances to the next packet. Otherwise, the packet remains unchanged, the attempted rule is added to T_t, and the next state used in the update is $s' = s_t$.

Periodically, the learned Q-values are converted into deployable rule orderings for the REE. In the single-state case, the optimizer produces one global ordering by sorting all rules in decreasing order of their learned value. In the general multi-state case, the optimizer instead produces one ordering for each traffic state:

$$\pi_{\text{ord}}(s) = \big(r_s^{(1)}, r_s^{(2)}, \ldots, r_s^{(N)}\big), \tag{6}$$

where $Q(s, r_s^{(1)}) \geq Q(s, r_s^{(2)}) \geq \cdots \geq Q(s, r_s^{(N)})$ and $\{r_s^{(1)}, \ldots, r_s^{(N)}\} = R$. Thus, $\pi_{\text{ord}}(s)$ is the ordering that ranks first the rules currently estimated as most promising for packets belonging to state s.

Algorithm 1: DTO learning procedure

Input: Rule set R, packet stream (p_t), abstraction ϕ, parameters α, γ, ϵ, update interval U
Output: State-conditioned rule orderings $\pi_{\text{ord}}(s)$ for all $s \in S$

```
Initialize Q(s, r) ← 0, for all s ∈ S and r ∈ R;
foreach packet p_t do
    s_t ← φ(p_t);
    T_t ← ∅;
    h ← 1;
    done ← false;
    while ¬done do
        A_{t,h} ← R \ T_t;
        Select r_{t,h} ∈ A_{t,h} according to the masked ε-greedy policy in (4);
        matched ← match(p_t, r_{t,h});
        ℓ_{t,h} ← |T_t|;
        Compute ρ_{t,h} according to (3);
        if matched then
            if a next packet p_{t+1} exists then
                s' ← φ(p_{t+1});
            else
                s' ← end;
            done ← true;
        else
            T_t ← T_t ∪ {r_{t,h}};
            s' ← s_t;
        if s' = end then
            Q(s_t, r_{t,h}) ← Q(s_t, r_{t,h}) + α [ρ_{t,h} − Q(s_t, r_{t,h})];
        else
            Q(s_t, r_{t,h}) ← Q(s_t, r_{t,h}) + α [ρ_{t,h} + γ max_{r∈R} Q(s', r) − Q(s_t, r_{t,h})];
        h ← h + 1;
    if t mod U = 0 then
        foreach s ∈ S do
            Build π_ord(s) by sorting all rules in R in descending order of Q(s, ·);
        Export {π_ord(s)}_{s∈S} to the REE;
```

4.6. TwinWall learning procedure

The overall learning procedure executed by the DTO is reported in Algorithm 1 and can be summarized as follows.

First, the Q-table is initialized, typically by assigning zero value to every state–rule pair. Then, as packets are observed from the incoming traffic stream, each packet is mapped to its abstract traffic state through ϕ. For the current packet, the optimizer repeatedly selects candidate rules according to the masked ϵ-greedy policy, evaluates them through the matching predicate, computes the corresponding reward, and updates the Q-table through (5). This process continues until a matching rule is found for the packet, after which the optimizer moves to the next packet.

After a fixed update interval, the current Q-table is transformed into one or more ordered rule views according to (6), and the resulting ordering information is exported to the REE. In this way, the learning process remains continuous and data-driven, while the production-side firewall is periodically refreshed with rule priorities learned directly from experience.

Table 1
Firewall rules used in the illustrative example.

Rule	srcIP	dstIP	srcPort	dstPort	proto	action
r_1	10.0.0.0/24	172.16.0.10	*	22	TCP	allow
r_2	10.0.1.0/24	172.16.0.20	*	80	TCP	allow
r_3	192.168.1.0/24	8.8.8.8	*	53	UDP	deny
r_4	192.168.2.0/24	172.16.0.30	*	5000	UDP	allow
r_5	10.0.2.0/24	172.16.0.40	*	443	TCP	allow

4.7. Memory and computational complexity

The computational and memory costs of TwinWall can be analyzed according to its three main processing stages: (i) DTO learning, (ii) generation of the state-conditioned rule orderings, and (iii) packet enforcement in the REE.

During the *DTO learning stage*, the main persistent data structure is the Q-table, which contains one value for each state–rule pair and therefore requires $O(|S|N)$ memory. For each packet p_t, the DTO additionally maintains the temporary set T_t of rules already evaluated unsuccessfully, whose size is at most N. Hence, excluding the storage required for the input traffic and firewall policy, the overall memory complexity of the learning stage is $O(|S|N)$. Concerning computation, at most N rules can be evaluated before a matching rule is found. With the adopted masked ϵ-greedy strategy, selecting the next admissible rule may require inspecting up to N Q-values. Since this selection can be repeated up to N times for the same packet, the worst-case computational cost of the DTO rule-selection procedure is $O(N^2)$ per packet. In practice, the actual cost depends on how early a matching rule is found, since the interaction terminates as soon as packet classification is completed.

During the *rule-order generation stage*, performed every U processed packets, the DTO converts the learned Q-values into one rule ordering for each traffic state. For each $s \in S$, the N rules are sorted according to their Q-values, resulting in a computational cost of $O(|S|N \log N)$ for each policy-update event. Maintaining all state-conditioned rule orderings requires $O(|S|N)$ memory.

Finally, during the *REE enforcement stage*, the REE selects the rule ordering associated with the current traffic state and performs ordinary sequential first-match packet classification. Since at most N rules need to be inspected, the worst-case packet-classification complexity remains $O(N)$, as in a conventional sequential first-match firewall. The DTO learning and rule-order generation costs are incurred outside the production packet-processing path and therefore do not alter the asymptotic classification complexity of the REE.

4.8. Clarifying example

To clarify the interaction between state abstraction, rule selection, Q-value updates, and rule-ordering extraction, a small illustrative example is provided here.

The firewall rule set considered in this example is reported in Table 1. For the sake of simplicity, let the state-abstraction function ϕ partition packets according to transport protocol only, so that $S = \{\text{TCP}, \text{UDP}, \text{OTHER}\}$. The Q-table is initially zero for every state–rule pair, and the reward parameters are defined such that $r^+ = 1.0$, $r^- = -0.1$, and $\lambda = 0.02$.

Two packets are sequentially analyzed by the DTO: $p_1 = (10.0.1.15, 172.16.0.20, 41000, 80, \text{TCP})$ and $p_2 = (192.168.1.25, 8.8.8.8, 53000, 53, \text{UDP})$.

Concerning the first packet, $\phi(p_1) = \text{TCP}$. At the beginning of its processing, the DTO has state $s_1 = \text{TCP}$ and $T_1 = \emptyset$.

At step $h = 1$, let the masked ϵ-greedy policy select rule $r_{1,1} = r_1$. Since r_1 does not match p_1, the reward is $\rho_{1,1} = r^- - \lambda \ell_{1,1} = -0.1 - 0.02 \cdot 0 = -0.1$, because no failed rule has been tested before this step, i.e., $\ell_{1,1} = 0$. The attempted rule is added to T_1, so that $T_1 = \{r_1\}$, and the interaction continues on the same packet.

Table 2
Learned Q-table.

State s	$Q(s, r_1)$	$Q(s, r_2)$	$Q(s, r_3)$	$Q(s, r_4)$	$Q(s, r_5)$
TCP	−0.0010	0.0098	0.0000	0.0000	0.0000
UDP	0.0000	0.0000	0.0100	0.0000	0.0000
OTHER	0.0000	0.0000	0.0000	0.0000	0.0000

At step $h = 2$, assume the optimizer selects rule $r_{1,2} = r_2$. This time the rule matches p_1, hence $\rho_{1,2} = r^+ - \lambda \ell_{1,2} = 1.0 - 0.02 \cdot 1 = 0.98$, because one unsuccessful attempt has already occurred, i.e., $\ell_{1,2} = 1$. The search for p_1 then terminates and the interaction advances to the next packet.

Concerning the second packet, $\phi(p_2) = \text{UDP}$. Let us assume that, at step $h = 1$, the selected rule is directly $r_{2,1} = r_3$. Since the rule matches immediately, the reward is $\rho_{2,1} = r^+ - \lambda \ell_{2,1} = 1.0 - 0.02 \cdot 0 = 1.0$, because $\ell_{2,1} = 0$.

After these interactions, the Q-table is updated so that successful rules receive higher values in the states where they are useful, while unsuccessful early attempts receive lower values. Assuming zero initialization and using $\alpha = 0.01$ and $\gamma = 0.9$, the Q-values after the above interactions take the form shown in Table 2.

From this Q-table, TwinWall derives one rule ordering for each state by sorting rules in descending order of their Q-values. In this example, the resulting orderings are therefore headed by the rules that were reinforced by the observed interactions, i.e., r_2 for TCP traffic and r_3 for UDP traffic. For TCP traffic, the rules with zero Q-values precede r_1, whose Q-value is negative, while rules with equal Q-values may appear in any order. For instance, one admissible set of orderings is:

$$\pi_{\text{ord}}(\text{TCP}) = (r_2, r_3, r_4, r_5, r_1),$$
$$\pi_{\text{ord}}(\text{UDP}) = (r_3, r_1, r_2, r_4, r_5),$$
$$\pi_{\text{ord}}(\text{OTHER}) = (r_1, r_2, r_3, r_4, r_5).$$

The extracted orderings can then be exported to the REE, where they are used as state-conditioned rule views for subsequent packet processing.

5. Implementation and validation

A prototype implementation of the TwinWall architecture was developed in Python 3 to assess the effectiveness of the designed reinforcement-learning algorithm. Three datasets were specifically selected to evaluate TwinWall under complementary traffic and validation conditions (Section 5.1). The first two consist of real packet traces collected in substantially different operational environments: a heterogeneous Internet-backbone setting and a specialized industrial-control network. These traces expose TwinWall to naturally occurring traffic patterns from two distinct application domains. The third dataset is instead a synthetic workload generated with the ClassBench toolchain, and provides a controlled and reproducible packet-classification benchmark with an explicitly generated firewall policy and traffic trace.

The experimental validation campaign was then carried out in two phases. In the first phase, multiple TwinWall configurations were evaluated in order to identify the best-performing setup in terms of packet processing, measured through misses and packet classification time, and the scalability of the selected configuration was assessed by increasing the firewall rule-set size (Section 5.2). In the second phase, the best-performing TwinWall configuration was compared with baseline rule-ordering strategies (Section 5.3). Finally, a comparative analysis with related approaches was conducted to discuss differences in computational workflow, deployment requirements, and performance (Section 5.4). All the tests were conducted on a workstation equipped with a 20-core Intel Core Ultra 7 265 processor, with a maximum frequency of 5.30 GHz, and 64 GB RAM.

5.1. Validation datasets

Three datasets with different characteristics were selected to validate TwinWall under heterogeneous traffic conditions. Two datasets consist of real packet traces collected in backbone and industrial-control environments, whereas the third one is a synthetic benchmark workload generated with the ClassBench toolchain. The former allow TwinWall to be evaluated on traffic observed in operational contexts, while the latter provides a controlled and reproducible setting based on a standard packet-classification benchmark.

5.1.1. MAWI and SCADA/ICS datasets

The first dataset consists of 300000 packets extracted from the MAWI traffic archive [27], specifically from the trace named `samplepoint-F` and recorded on January 1st, 2026 in the file `202601011400.pcap`. This trace captures real backbone traffic and therefore exposes TwinWall to a heterogeneous mix of endpoints, protocols, and port combinations. Its main value for validation lies in the diversity of the observed traffic, which makes it suitable for evaluating TwinWall on a compact Internet-facing workload characterized by broad variability in communication patterns.

The second dataset consists of 239267 packets extracted from the *SCADA/ICS Network Captures* at 4SICS [28], specifically from the trace recorded in the file `4SICS-Geek- Lounge-151020.pcap`. Unlike the MAWI subset, this dataset represents traffic generated in an industrial-control environment, where communications are tied to specific devices and operational services. As a consequence, it captures a markedly different traffic setting, with patterns that reflect structured interactions among field devices, hosts, and control-related services rather than general-purpose backbone activity. The trace still includes multiple transport protocols, with TCP as the dominant one and smaller shares of UDP and ICMP traffic, thus preserving protocol-level diversity within a domain-specific operational context.

The two real traces therefore represent substantially different operational contexts. MAWI captures heterogeneous backbone traffic from a general Internet setting, whereas the SCADA/ICS trace captures communication patterns associated with a specialized industrial environment. This distinction allows the real-traffic validation to cover both broad general-purpose communication patterns and more structured domain-specific behavior.

For each of these two real-traffic datasets, a firewall policy composed of 500 rules was generated from the observed packet trace. This generation process was specifically implemented to ensure that the resulting rules are coherent with the underlying real traffic, rather than being produced independently from it. The generator adopts a K-Means-based approach that extracts recurrent patterns from packet data and translates them into filtering rules. The overall process is structured into five main phases.

Phase 1: Protocol-wise partitioning. The input packet trace is first partitioned by protocol, and each subset is processed independently in the next steps. This design avoids mixing traffic classes with different semantics and improves the coherence of the generated rules. The total budget of 500 rules to be generated is distributed proportionally to the traffic share of each protocol, so that more frequent protocols receive a larger number of clusters and, consequently, more rules.

Phase 2: Feature extraction and preprocessing. Each packet is represented with its 5-tuple $(srcIP, dstIP, srcPort, dstPort, protocol)$, which is transformed into a numerical feature vector suitable for clustering. IP addresses are encoded using an adaptive hierarchical representation, whose granularity depends on the observed diversity of addresses. For TCP and UDP traffic, additional port-related features are included, such as normalized source and destination ports, categorical encodings of IANA port classes, and simple relational features between ports. All features are then standardized to ensure that no attribute dominates the distance computation.

Phase 3: K-means clustering. Clustering is performed independently for each protocol-specific subset. The number of clusters assigned to each subset follows the proportional allocation defined in Phase 1. K-Means is applied with multiple initializations to improve robustness against centroid placement. Each cluster identifies a group of packets sharing similar 5-tuple characteristics, effectively capturing recurrent communication patterns present in the trace.

Phase 4: Rule synthesis. Each cluster is translated into a firewall rule by generalizing the packets it contains. In particular, source and destination IP addresses are converted into covering ranges, source and destination ports are expressed as intervals, and the protocol field is simply preserved as it is. In this way, each cluster yields one rule that compactly summarizes a set of similar packets observed in the dataset.

Phase 5: Controlled rule overlap. To obtain more realistic rule sets, the implementation also introduces a controlled amount of overlap among some of them, in order to obtain rule sets that are more realistic for policy-reordering experiments. In particular, part of the generated rules are turned into partially overlapping rules by splitting one ranged field into two intersecting subranges, while another fraction is transformed into fully overlapping pairs by creating one more specific rule entirely contained within a more general one.

The resulting MAWI and SCADA/ICS rule sets each consist of 500 rules that are derived from the corresponding observed traffic, maintain consistency with its packet distribution, and include controlled overlap patterns.

5.1.2. ClassBench dataset

The third dataset was generated using the ClassBench toolchain, a widely adopted benchmark framework for evaluating packet-classification algorithms [29,30]. Unlike the MAWI and SCADA/ICS datasets, in this case both the firewall rule set and the corresponding packet trace were generated synthetically.

First, ClassBench-ng [31], an extension of the original ClassBench framework that supports the characteristics of modern packet-classification rule sets, was used to generate a packet-filtering rule set composed of 500 rules in FW rule format. The generated policy specifies matching conditions over the packet-header fields considered by TwinWall and was directly used by both the DTO and the REE.

Then, the corresponding packet trace, composed of 220009 packets, was produced through the Trace Generator distributed with the original ClassBench implementation [32]. The original generator was adopted because the publicly available ClassBench-ng source code does not include a trace-generation component, while the rule sets produced by ClassBench-ng remain compatible with the input format expected by the original ClassBench Trace Generator. The resulting trace follows a Pareto distribution, so that a limited subset of rules accounts for a larger fraction of packet matches, while the remaining rules are exercised less frequently. This workload therefore complements the two real traces by providing a controlled rule-popularity distribution based on a standard packet-classification benchmark.

5.1.3. Complementarity of the validation datasets

The three datasets provide complementary perspectives on the behavior of TwinWall. MAWI evaluates the approach under heterogeneous backbone traffic with broad variability in endpoints, protocols, and communication patterns; 4SICS evaluates it in a more structured and domain-specific industrial-control environment; and ClassBench provides a controlled and reproducible workload in which the firewall policy and packet trace are generated through a standard benchmarking toolchain. These differences are relevant for adaptive rule ordering because they expose the learning mechanism to substantially different traffic structures and rule-activation patterns. The three workloads are therefore intended to provide heterogeneous validation conditions rather than multiple instances of the same traffic scenario.

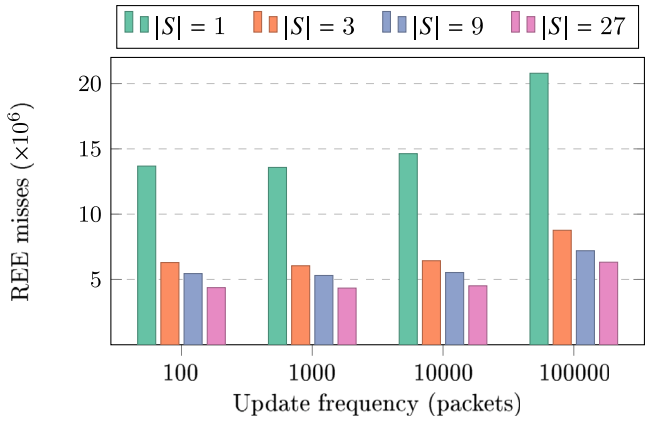


Fig. 2. REE misses on MAWI dataset without pre-training.

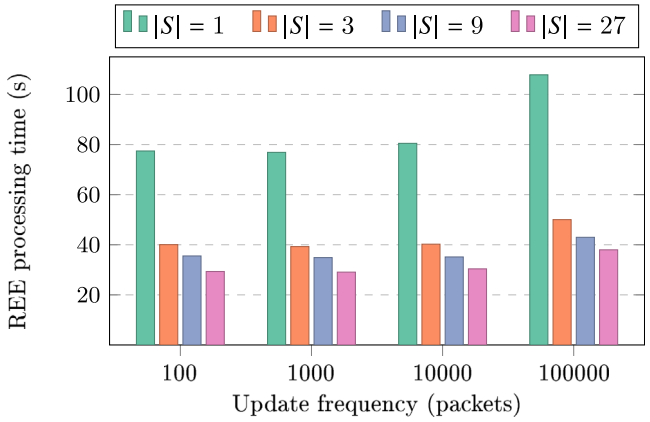


Fig. 3. REE processing time on MAWI dataset without pre-training.

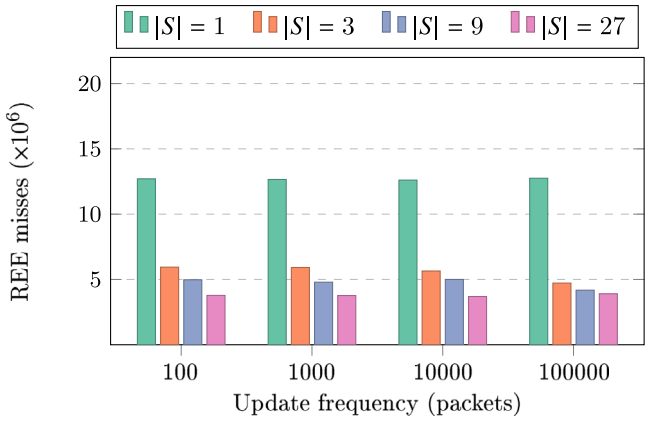


Fig. 4. REE misses on MAWI dataset with 3-epoch pre-training.

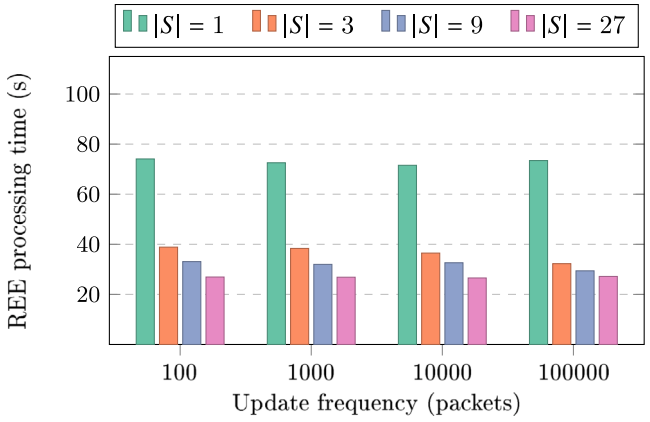


Fig. 5. REE processing time on MAWI dataset with 3-epoch pre-training.

5.2. Packet processing efficiency validation

The TwinWall prototype was applied to the three datasets to validate its packet processing efficiency under different operating conditions. Since TwinWall operates as a digital-twin architecture, the DTO and the REE were fed with the same packet stream in all experiments, and rule-order updates were transferred from the former to the latter at fixed intervals. Since TwinWall operates by reordering the rules of a conflict-free policy, it does not modify the filtering rules or their associated actions; therefore, for the policies considered in this study, the final filtering decision for a given packet is preserved across admissible rule orderings. Accordingly, the tests aimed to assess the practical quality of the rule orderings periodically exported by the DTO to the REE, and therefore were based on two metrics: *REE misses*,[2] i.e., the number of rule checks performed by the REE before reaching the matching rule, and *REE processing time*, i.e., the time spent by the REE to process the validation stream.

Two different initialization setups were considered for this test campaign. In the first setup, the DTO starts without pre-training, the REE starts with a random rule ordering, and the system must adapt online from scratch as packets are observed. This setup models a cold-start condition, where no prior knowledge of the traffic distribution is assumed. In the second setup, the DTO performs 3 epochs of pre-training before validation. As one epoch denotes one complete pass over the whole packet trace used for that dataset, 3 epochs correspond to three full preliminary passes over the trace before the actual validation run starts. This setup models a warm-start condition, in which the DTO is allowed to stabilize an initial rule ordering before the REE begins processing the validation traffic. Comparing these two cases makes it possible to separate the impact of online adaptation alone from the benefit of a prior traffic-aware initialization.

Two other setup parameters that were varied across the experiments are the *number of states* used by the reinforcement-learning algorithm of the DTO and the *update frequency*.

Concerning the number of states $|S|$, the tested values were $\{1, 3, 9, 27\}$. The 1-state configuration corresponds to a non-contextual agent that learns a single global preference over the rules, the 3-state configuration separates packets by transport protocol, the 9-state configuration refines this representation by combining protocol information with source-port classes, and the 27-state configuration further distinguishes both source- and destination-port classes. These configurations were selected to obtain a controlled and interpretable progression of the state-space granularity while relying only on header fields that are natively available to the considered packet-filtering firewall. Protocol and port categories provide lightweight traffic descriptors that can be extracted independently for each packet, without requiring flow reconstruction, session tracking, application identification, temporal aggregation, or additional external information. This choice also makes it possible to isolate the effect of increasing contextual granularity, since each configuration incrementally extends the preceding one while preserving the same learning algorithm and experimental conditions. As defined in Section 4.2, the TwinWall

[2] In this paper, the term REE misses specifically denotes unsuccessful rule evaluations performed before reaching the matching rule, and should not be interpreted as an incorrect filtering decision.

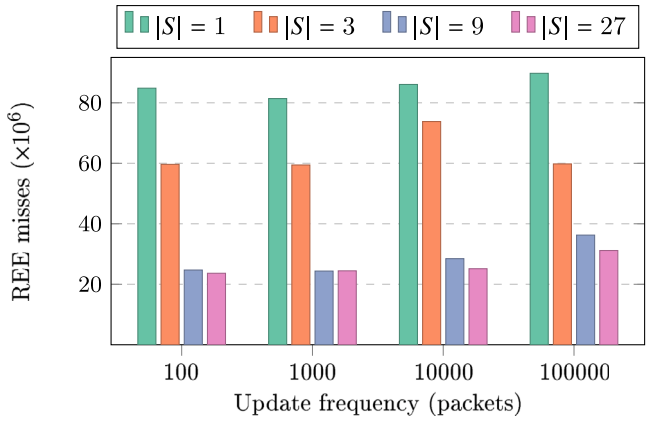


Fig. 6. REE misses on 4SICS dataset without pre-training.

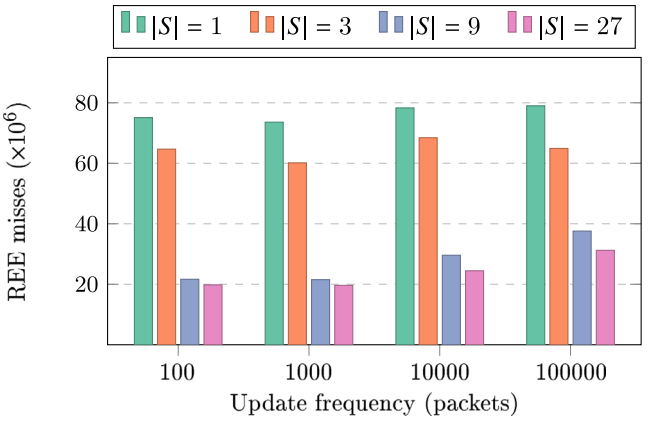


Fig. 8. REE misses on 4SICS dataset with 3-epoch pre-training.

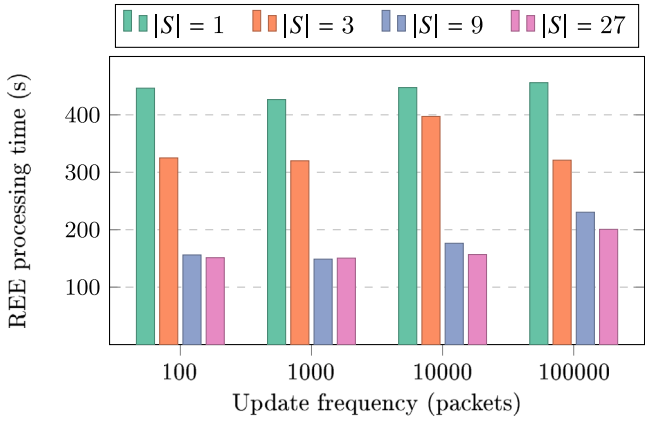


Fig. 7. REE processing time on 4SICS dataset without pre-training.

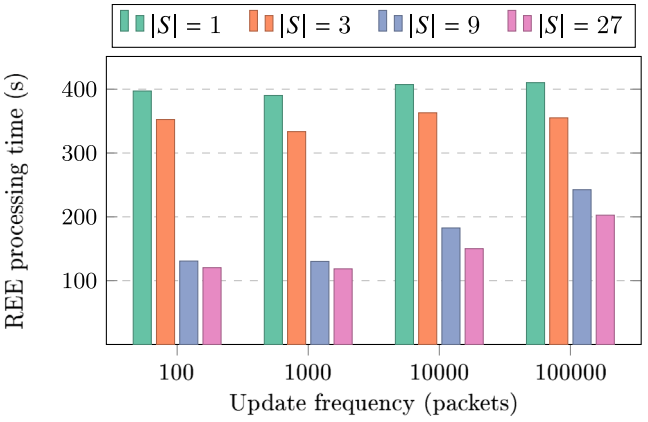


Fig. 9. REE processing time on 4SICS dataset with 3-epoch pre-training.

model could also support abstraction functions based on other traffic-level characteristics, such as session context and flow statistics. Such richer abstractions were not considered in this validation because they would introduce additional design choices and processing mechanisms, making it more difficult to assess the effect of state-space granularity in isolation and increasing both the size of the state space and the amount of experience required to learn reliable state-conditioned priorities.

Concerning the update frequency, expressed as the number of packets processed by the DTO between two consecutive rule-order exports to the REE, the tested values were $\{100, 1000, 10000, 100000\}$ packets. Small values correspond to very frequent refreshes of the REE rule order, whereas large values leave the REE operating longer with a fixed ordering. This parameter is therefore important to quantify the trade-off between fast adaptation and stale rule tables.

All tests used the same learning parameters across all datasets and across all combinations of initialization method, number of states, and update frequency. During DTO pre-training, when performed, the Q-learning procedure used learning rate $\alpha = 0.01$, discount factor $\gamma = 0.9$, initial exploration rate $\epsilon = 0.1$, and multiplicative exploration decay equal to 0.995 after each packet update. During the actual validation, the DTO continued to update its table online with a more conservative configuration, using $\epsilon = 0.02$ and a learning rate reduced to 0.003. The reward model assigned +1.0 to a successful match and −0.1 to a failed rule evaluation, with an additional penalty of −0.02 for each rule already attempted on the same packet.

These parameter values were selected according to the role of each parameter in the learning process and were kept fixed across all experiments. During pre-training, the learning rate $\alpha = 0.01$ enables gradual Q-value updates, while $\gamma = 0.9$ assigns substantial weight to the utility of subsequent rule selections within the packet-classification process. The initial exploration rate $\epsilon = 0.1$, combined with its multiplicative decay, allows the agent to explore alternative rules during the early learning phase and progressively favor exploitation as its estimates become more stable. During validation, the lower values of $\epsilon = 0.02$ and $\alpha = 0.003$ limit ordering fluctuations while still allowing adaptation to the observed traffic. Finally, the reward values were chosen to make a successful match clearly preferable to an unsuccessful proposal, while the incremental penalty discourages long rule-search chains.

The results of this validation campaign are reported in Figs. 2–13, which show REE misses and REE processing time on the three datasets under different numbers of states, update frequencies, and pre-training settings. Four main observations emerge from these results.

First, increasing the number of states improves performance in most configurations across all datasets. The improvement is visible not only when moving from the stateless configuration ($|S| = 1$) to the first context-aware one, but also when adopting finer-grained state spaces such as $|S| = 9$ and $|S| = 27$. This trend is reflected in both metrics: REE misses decrease because the learned ordering places the matching rules earlier more often, and REE processing time decreases accordingly because fewer rule evaluations are needed on average. The effect is already clear on the MAWI dataset, but it becomes substantially stronger on the 4SICS dataset, where the gap between the 1-state and 27-state configurations is much wider. A similar overall benefit is observed on the ClassBench workload, although, without pre-training, the 9-state configuration does not always outperform the coarser alternatives. For this dataset, partitioning traffic according to protocol and source-port class alone results in an unfavorable distribution of packets across the states, so that some state-conditioned orderings are learned from

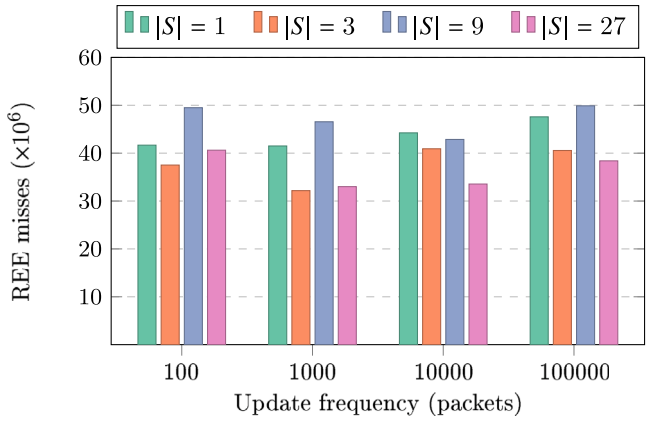


Fig. 10. REE misses on ClassBench dataset without pre-training.

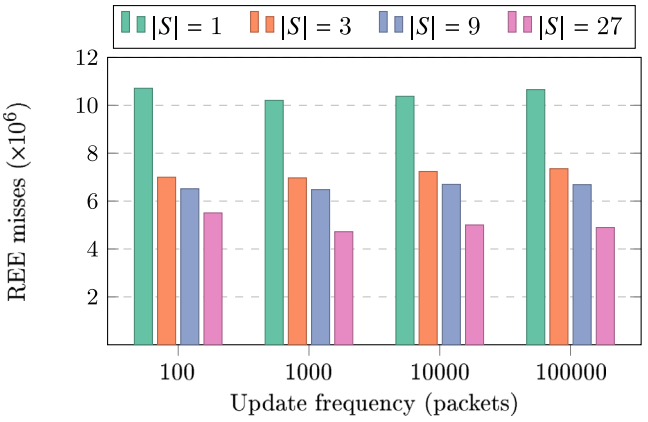


Fig. 12. REE misses on ClassBench dataset with 3-epoch pre-training.

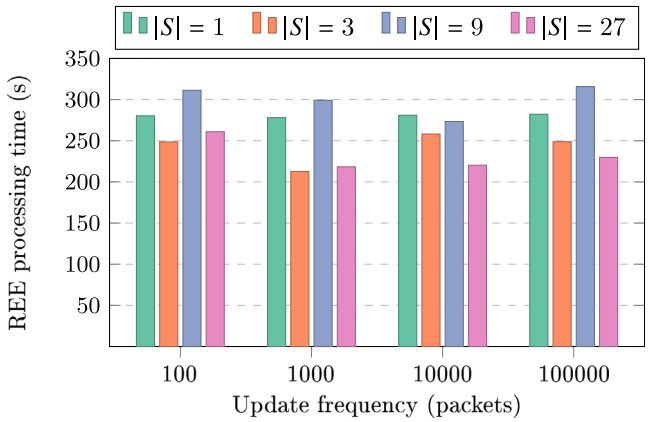


Fig. 11. REE processing time on ClassBench dataset without pre-training.

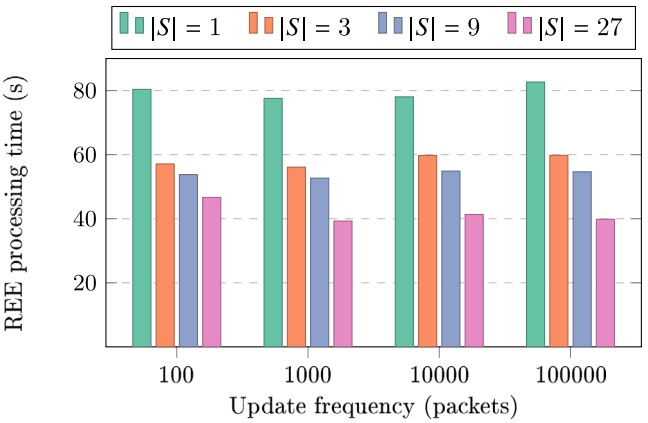


Fig. 13. REE processing time on ClassBench dataset with 3-epoch pre-training.

limited experience. However, after three pre-training epochs, this effect is overcome and the 9-state configuration again outperforms the 1-state and 3-state ones. Therefore, this first result overall shows that richer state abstractions are commonly more beneficial.

Second, the impact of the update frequency varied depending on the initialization setup. In the configurations without pre-training, more frequent updates generally produce better results, since the DTO starts from a largely uninformed rule ordering and must learn online from scratch. In this condition, exporting updated rule orders more often allows the REE to benefit earlier from the information progressively accumulated during validation. By contrast, in the 3-epoch configurations, the best results are in most cases obtained with an update frequency of 1000 packets, rather than with the most aggressive setting of 100 packets. Once pre-training has already produced a reasonably good initial ordering, the role of online updates changes: the objective is no longer to correct a poor initial condition as quickly as possible, but rather to refine an already meaningful ordering while preserving stability. If updates are performed too frequently, the exported rule order may overreact to short-term traffic fluctuations that are not representative of the broader packet distribution, causing unnecessary oscillations in the REE configuration. An update frequency of 1000 packets appears to provide a better trade-off between responsiveness and robustness, remaining frequent enough to follow traffic evolution while avoiding excessive sensitivity to transient local patterns.

Third, pre-training has a positive impact in almost all configurations. The 3-epoch setup consistently improves over the 0-epoch one in terms of both REE misses and REE processing time, showing that allowing the DTO to perform a small number of preliminary passes over the trace leads to a better starting policy. This advantage is not limited to the very beginning of validation, but propagates throughout the execution because the REE starts from a more effective ordering and therefore incurs a lower cumulative filtering cost. The benefit is present on all three datasets, but it is more evident on the 4SICS dataset, where the larger traffic heterogeneity amplifies the cost of poor early decisions and makes a better initialization more valuable, and even more on the ClassBench dataset, where the Pareto-distributed traffic concentrates matches on a subset of rules and pre-training allows the DTO to identify these dominant matching patterns before validation begins.

Fourth, the comparison among the three datasets is coherent with their different traffic characteristics and further clarifies the operating conditions under which TwinWall is most beneficial. The MAWI dataset exhibits a more regular and compact behavior: although the advantages of larger state spaces, pre-training, and appropriate update frequencies are clearly visible, the performance differences among configurations remain relatively contained. By contrast, the 4SICS dataset is globally more demanding, both in terms of REE misses and processing time, and shows a stronger sensitivity to the TwinWall configuration. The ClassBench dataset further emphasizes the importance of initialization and traffic-aware adaptation, since its Pareto-distributed rule popularity makes pre-training particularly effective in identifying the rules responsible for most packet matches. This analysis suggests that, in more complex traces, the cost of suboptimal rule ordering accumulates more rapidly, making both a good initialization and a sufficiently expressive state abstraction more important.

Overall, these results show that TwinWall is most effective in improving firewall packet processing efficiency when it combines three elements: a sufficiently rich state representation, some pre-training, and periodic, but not excessively aggressive, rule-order updates. Richer

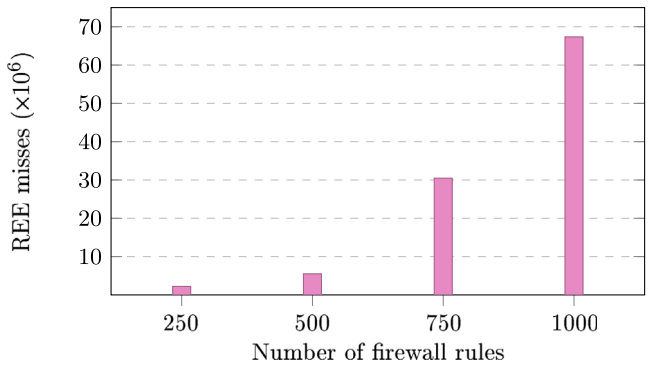


Fig. 14. REE misses for increasing ClassBench rule-set sizes.

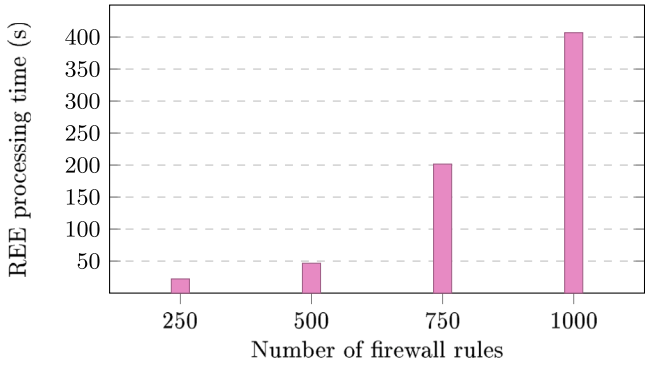


Fig. 15. REE processing time for increasing ClassBench rule-set sizes.

state spaces allow the DTO to capture traffic regularities more accurately, pre-training provides a better starting ordering for the REE, and online updates preserve this advantage as the packet stream evolves.

5.2.1. Scalability evaluation

To assess the behavior of TwinWall with increasing firewall-policy sizes, an additional scalability experiment was conducted using the ClassBench toolchain. Following the same procedure described in Section 5.1, four firewall rule sets composed of 250, 500, 750, and 1000 rules were generated, together with a corresponding trace of one million packets for each rule set. The best-performing TwinWall configuration identified in the previous experiments, i.e., the setup with 3 pre-training epochs, update frequency equal to 1000 packets, and $|S| = 27$, was used in all cases, so that the observed variations could be attributed primarily to the increasing number of rules.

Figs. 14 and 15 report the resulting REE misses and processing times. As expected for a sequential first-match firewall, both metrics increase with the rule-set size. The average number of unsuccessful rule evaluations (REE misses) per packet increases from approximately 2.22 with 250 rules to 5.51, 30.47, and 67.35 with 500, 750, and 1000 rules, respectively. The corresponding average processing time per packet increases from approximately 22.10 µs to 46.64 µs, 201.53 µs, and 406.76 µs. The growth becomes more pronounced for the two largest rule sets, indicating that larger action spaces require more packet observations to learn sufficiently effective rule priorities and that any remaining suboptimal placement has a greater impact on sequential matching. Nevertheless, the experiment confirms that TwinWall can operate with rule-set sizes consistent with operational firewall deployments.

5.2.2. DTO resource utilization

In addition to packet-processing scalability, we evaluated the memory requirements introduced by the DTO learning component. Since the learned representation is stored as a Q-table containing one value for each state–rule pair, its memory footprint depends directly on the number of traffic states and firewall rules. To characterize this cost independently of Python runtime and library overhead, we report the exact storage required by the Q-table representation used by the prototype. The experiments use the same $|S| = 27$ configuration adopted in the scalability evaluation, with Q-values stored as double-precision floating-point values.

Table 3
Memory requirements of the DTO Q-table for increasing firewall-policy sizes with $|S| = 27$.

Rules	Q-table entries	Memory (KiB)
250	6750	52.73
500	13,500	105.47
750	20,250	158.20
1000	27,000	210.94

As shown in Table 3, the memory required by the learned representation increases linearly with the firewall-policy size. For $|S| = 27$, the Q-table requires 52.73 KiB with 250 rules, 105.47 KiB with 500 rules, 158.20 KiB with 750 rules, and 210.94 KiB with 1000 rules. These results are consistent with the $O(|S|N)$ memory complexity derived in Section 4.7. Even for the largest evaluated policy, the learned representation requires approximately 211 KiB, showing that the memory overhead associated with the DTO Q-table remains limited within the considered experimental range.

5.3. Comparison with baseline strategies

Then, the prototype implementation was evaluated in comparison with three non-reinforcement-learning strategies:

- *Static Firewall*, i.e., a traditional first-match firewall with a fixed rule order, where no learning or reordering is performed during execution;
- *Offline Frequency Ordering*, in which rules are sorted once before validation according to how frequently they match during a preliminary training phase, and then kept fixed throughout execution;
- *Window Frequency Reordering*, in which the rule order is periodically recomputed according to the match frequencies observed in the most recent update window, thus providing a dynamic baseline without reinforcement learning.

For this comparison, TwinWall was again represented by its best-performing configuration emerging from the previous validation, i.e., the setup with 3 pre-training epochs, update frequency equal to 1000 packets, and $|S| = 27$. The results are reported in Figs. 16–21, separately for the three datasets and for the two evaluation metrics (REE misses and processing time). The comparison highlights that TwinWall outperforms all the considered baselines on all three datasets, both in terms of REE misses and in terms of REE processing time.

On the MAWI and ClassBench datasets, the three reference strategies follow an intuitively progressive behavior: the static firewall is the least effective one, offline frequency ordering provides a clear improvement, and window-based frequency reordering performs better still. This outcome suggests that, in these datasets, increasingly adaptive frequency-based strategies are able to exploit the traffic regularities more effectively. In this perspective, TwinWall achieves a substantially better efficiency than all of them. On the MAWI dataset, it reduces REE misses to about 3.8×10^6 and REE processing time to about 26.9 s; compared with the best non-reinforcement-learning baseline, this corresponds to reductions by factors of approximately 4.6 and 3.5, respectively. On the ClassBench dataset, TwinWall reduces REE misses to about 4.7×10^6 and REE processing time to about 39.3 s, corresponding to reductions by factors of approximately 8.2 and 6.3 with respect to the best non-reinforcement-learning baseline. In both cases, the gap becomes even larger with respect to the static baseline.

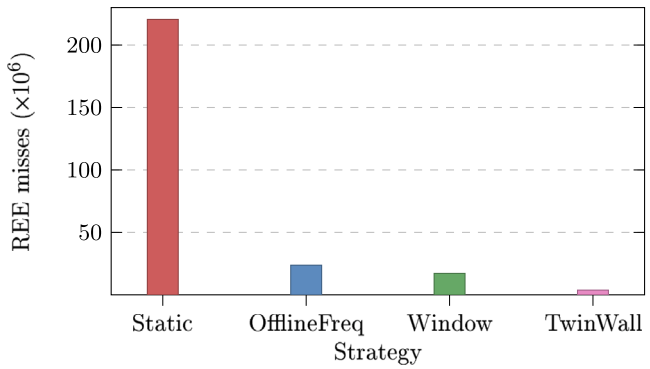


Fig. 16. Comparison of REE misses on MAWI dataset.

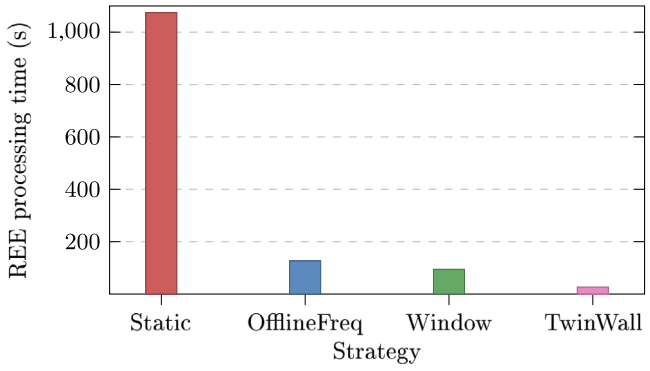


Fig. 17. Comparison of REE processing time on MAWI dataset.

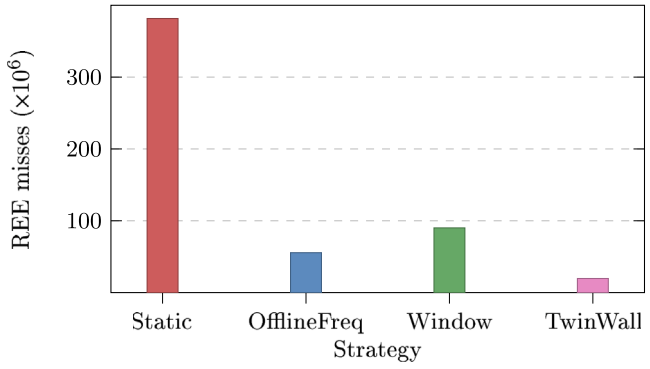


Fig. 18. Comparison of REE misses on 4SICS dataset.

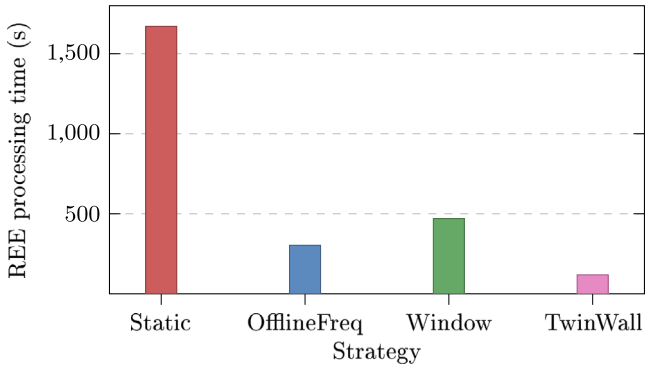


Fig. 19. Comparison of REE processing time on 4SICS dataset.

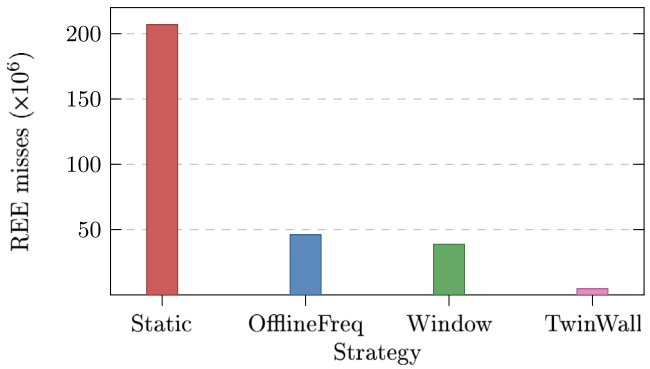


Fig. 20. Comparison of REE misses on ClassBench dataset.

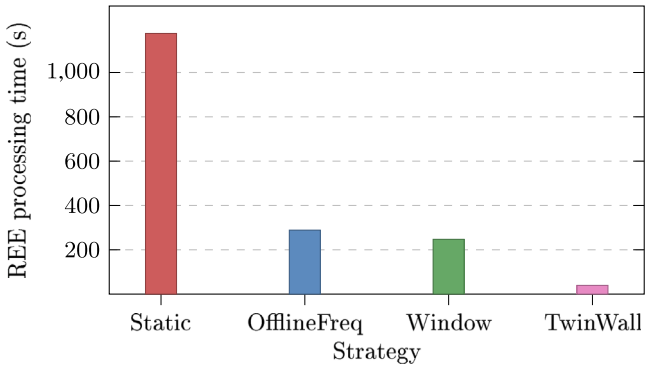


Fig. 21. Comparison of REE processing time on ClassBench dataset.

The same general conclusion holds on the 4SICS dataset, but with a qualitatively different ordering among the non-reinforcement-learning baselines. In this case, Offline Frequency Ordering performs better than Window Frequency Reordering, reversing the trend observed on the MAWI dataset. The reason is that in this trace a simple window-based adaptation can be less effective than a stable ordering computed over a broader traffic sample, likely because short-term frequency fluctuations are not sufficiently representative of the longer-term traffic structure. TwinWall still remains the best-performing strategy overall. It reduces REE misses to about 19.6×10^6 and REE processing time to about 118.5 s, which corresponds to an improvement of almost a factor of 3 in misses and more than a factor of 2.5 in processing time with respect to the best non-reinforcement-learning baseline, and to a much larger gain with respect to the static firewall.

Overall, these results show that the advantage of TwinWall does not merely depend on performing dynamic reordering, since a dynamic frequency-based strategy is not consistently the strongest baseline across the two datasets. Rather, the benefit appears to come from the way rule priorities are learned. TwinWall is able to remain superior in all three scenarios, despite the different relative behavior of the non-reinforcement-learning strategies, which suggests that reinforcement-learning-driven adaptation provides a more robust way to combine responsiveness and stability across heterogeneous traffic conditions.

5.4. Comparative analysis with other rule-reordering approaches

Table 4 summarizes the main approaches discussed in Section 2, highlighting the addressed problem, the adopted optimization technique, the required traffic information and update mode, the operational integration, and representative results reported in the original papers. In particular, beyond the specific optimization algorithm, the table shows how the considered approaches differ from TwinWall from two additional perspectives: the computational workflow required to

Table 4
Comparison of TwinWall with firewall packet-classification and rule-ordering optimization approaches.

Ref.	Problem	Technique	Traffic information and update mode	Operational integration	Representative results
Static and batch approaches					
[10]	ORO	DAG-based adjacent-rule sorting	Rule-matching probabilities; one-shot offline optimization	External optimizer producing one linear ordering	Up to approximately 70% fewer average comparisons than the initial rule order in the absence of dependencies.
[11]	ORO	Simulated annealing	Rule-matching probabilities; one-shot offline optimization	External optimizer producing one linear ordering	Up to approximately 5% fewer average comparisons than the heuristic in [10] under sparse precedence constraints.
[12]	ORO	Sub-Graph Merging (SGM)	Rule-matching probabilities; one-shot offline optimization	External optimizer producing one linear ordering	Up to approximately 8% lower matching cost than [9] and 12% lower than [10].
[13]	ORO	Dependency-aware insertion heuristic	Rule-matching probabilities; offline or on-demand optimization	External optimizer producing one linear ordering	Up to approximately 1.7%–28% lower matching cost than [9]; optimization completed in the order of 10 s.
[14]	ORO	Corrected SGM with adjacency lists	Rule weights from a packet distribution; one-shot offline optimization	External optimizer producing one linear ordering	Up to approximately 0.17% lower classification latency and 99.9% lower reordering time than [12].
[15]	ORO	Direct-dependent-rule heuristic	Rule weights from a packet distribution; one-shot offline optimization	External optimizer producing one linear ordering	Up to approximately 40% fewer packet–rule comparisons than [10].
[16]	RORO	ZDD-based heuristic	Packet-arrival distribution; one-shot offline optimization	External optimizer producing ordering-dependent rule weights	Up to approximately 18% lower classification latency than [10].
[17]	RORO	ZDD-based simulated annealing	Packet-arrival distribution; one-shot offline optimization	External optimizer using ZDDs for policy-equivalence verification	Up to approximately 11% lower classification latency than SGM [12].
[18]	RORO	Pairing of dependent rules	Rule weights from a packet distribution; one-shot offline optimization	External optimizer producing one linear ordering	Up to approximately 0.68% lower classification latency than SGM [12].
[19,20]	ORO	Genetic algorithm with topological constraint handling	Zipf-modeled rule-activation frequencies; one-shot offline optimization	External optimizer producing one precedence-preserving linear ordering	In the extended evaluation [20], cumulative displacement of 3.6% from the exact solution, with reordering times in the order of 10^{-1} s.
[21]	Skip-aware ordering for PF-like firewall scanners	Similarity-based jumpable-block construction	No traffic-distribution information required; one-shot offline optimization	External optimizer producing one linear ordering organized into skip-enabled blocks; directly deployable on PF-like scanners	Lowest packet–rule comparison count in all IPC experiments and in 50% of the ACL and FW experiments, i.e., in the ones based on rule sets with at least 2000 rules.
Dynamic approaches					
[9]	Dynamic ORO	Traffic-aware active-rule selection and reordering	Online matching-frequency and recency statistics; periodic or performance-triggered updates	Statistics collection and rule-list reconstruction integrated with the firewall	For a 200-rule policy evaluated on packet traces of unspecified size, up to approximately 60% fewer rule comparisons than the original unoptimized ordering, with 2–5 rule-list reconstructions per hour during peak periods.
[23]	Dynamic ORO with field reordering	Window-based traffic histograms	Online rule- and field-level histograms; periodic window-based updates	Runtime optimizer updating both rule and field orderings integrated with the firewall	For 500 rules and two million simulated packets, approximately 61% lower cumulative processing time than static ordering.
[24,25]	Dynamic ORO	Swapping-window heuristic	Online matching-probability estimates; periodic or performance-triggered updates	Runtime traffic estimator and rule-reordering component	For two 8-rule proof-of-concept policies, the average matching time decreased from 3.4921 to 2.6535 and from 5.1427 to 2.6619. On synthetic 100-rule policies, the method achieved average matching times 62.8% and 38.4% lower than [10], for dependency probabilities of 1% and 5%, respectively.
Alternative approaches					
[26]	Rule prediction	Decision-tree classifier	Labeled packet traces; offline training and per-packet prediction	Auxiliary classifier inserted before the sequential rule scan, with fallback on prediction failure	For a 13-rule policy evaluated on one million packets, cumulative processing time approximately 40% lower than static ordering.

derive and update the rule ordering, and the deployment requirements imposed on the target firewall architecture.

From the computational perspective, most static ORO and RORO methods execute a separate optimization procedure over a complete rule set and return one linear ordering. Their computational cost is therefore concentrated in the reordering phase and depends on the adopted search strategy and dependency representation. Dynamic approaches distribute part of this cost over time, but require runtime traffic estimators, update triggers, and repeated rule-list reconstruction. Instead, TwinWall follows a different workflow: the DTO continuously performs rule evaluations and Q-value updates on the observed traffic, while the REE performs only the state-based dispatch, when required, and the ordinary sequential first-match scan. Therefore, the learning overhead does not directly increase the processing cost of production packets. Naturally, this separation does not remove the computational cost of adaptation, as it transfers such cost to the DTO, whose memory requirements grow with the number of state–rule pairs and whose processing workload depends on the number of rule evaluations performed during learning. As quantified in Section 5.2.2, the memory required by the learned Q-table grows linearly with the policy size and

remains approximately 211 KiB for the largest evaluated configuration of 1000 rules and 27 traffic states. Although TwinWall consequently requires additional computational resources with respect to a standalone sequential firewall, these resources can be provided externally and remain independent of the production data path.

A useful contextual reference is provided by the evaluation of alternative approaches reported in [21], which also considered a 500-rule ClassBench firewall policy and approximately one million packets for the FW rule format. In that experiment, the evaluated ordering approaches required a number of packet–rule comparisons in the order of a few million. Under a different ClassBench-generated policy and packet trace of the same size, the best configuration of TwinWall required approximately 5.51×10^6 unsuccessful rule evaluations, corresponding to 5.51 REE misses per packet. These values do not provide a direct performance ranking, since the two evaluations use different generated policies and traffic instances. Nevertheless, as the benchmarks have comparable sizes, they show that TwinWall operates within the same order of magnitude as established approaches, while additionally supporting continuous traffic-driven adaptation.

From the deployment perspective, conventional batch optimizers can be executed only when a new ordering is required, whereas dynamic methods must be connected to the firewall runtime for collecting statistics and triggering policy updates. Methods such as FROG [21] and the rule-prediction approach [26] have additional requirements on the packet-classification procedure, respectively through skip-aware scanning support and an auxiliary classifier placed before the sequential rule scan. TwinWall instead preserves the conventional first-match enforcement procedure, but requires the deployment of the DTO as an additional component. In particular, the DTO must receive representative traffic observations, be allocated sufficient computational resources for learning, and interact with the firewall management plane to install complete rule orderings. As discussed in Section 3.3, it may be deployed externally or within an isolated environment on the firewall platform. This dependency on the digital twin increases the architectural and integration requirements with respect to a standalone firewall, but it is also the mechanism that prevents learning and exploration from affecting the latency of production packet processing.

6. Conclusions, limitations, and future work

This paper presented TwinWall, a digital-twin-based firewall architecture aimed at improving packet classification efficiency by decoupling the algorithm in charge of optimizing firewall rule reordering and the filtering procedure. The digital twin uses reinforcement learning to reward possible rule selections based on their impact on classification efficiency, and then periodically exports reordered firewall rules to the real firewall. In this way, TwinWall avoids embedding learning logic directly into the packet-processing path, thus keeping the operational firewall lightweight, while still benefiting from continuously refined rule orderings. The validation campaign conducted on the prototype implementation using two real packet traces and one synthetic ClassBench workload showed that the proposed design can effectively reduce both the number of rule checks required for packet classification and the corresponding packet processing time. The experiments also highlighted that the benefit of TwinWall becomes more evident when richer state abstractions, moderate pre-training, and sufficiently frequent, but not overly aggressive, rule updates are combined.

The present study still has some scope limitations that motivate further investigation. First, the current validation is based on a software prototype, and therefore does not yet capture the complete behavior of TwinWall when integrated with a production firewall platform. In particular, deployment-specific aspects such as traffic-observation overhead, communication with an external DTO, and policy-update latency could be evaluated end-to-end in such an integration scenario. Second, the current implementation adopts tabular Q-learning and has been experimentally assessed with a finite number of traffic states and firewall policies containing up to 1000 rules. Although the obtained scalability results are encouraging, larger policies and richer state representations may require additional learning mechanisms or implementation optimizations. Finally, the present formulation focuses on conflict-free sequential first-match policies, which allows rule priorities to be adapted without changing the filtering semantics. Extending the approach to policies with explicit rule dependencies represents an additional direction for future investigation.

These limitations naturally define several directions for future work. First, TwinWall will be integrated with real firewall platforms and high-speed packet-processing environments to evaluate the complete deployment workflow under production-level throughput and latency constraints. Second, richer state abstractions, possibly including additional packet- or flow-level features, will be investigated to improve the expressiveness of the learned rule-selection policies. In parallel, function approximation and more advanced reinforcement-learning techniques will be considered to support larger state spaces and firewall policies. Finally, future work will investigate dependency-aware extensions of TwinWall, so that adaptive rule prioritization can also be applied when reordering choices are constrained by inter-rule dependencies.

CRediT authorship contribution statement

Daniele Bringhenti: Writing – original draft, Validation, Methodology, Investigation, Conceptualization. **Fulvio Valenza:** Writing – review & editing, Supervision, Investigation, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

[1] A. Voronkov, L.H. Iwaya, L.A. Martucci, S. Lindskog, Systematic literature review on usability of firewall configuration, ACM Comput. Surv. 50 (6) (2018) 87:1–87:35, http://dx.doi.org/10.1145/3130876.

[2] E. Al-Shaer, H.H. Hamed, R. Boutaba, M. Hasan, Conflict classification and analysis of distributed firewall policies, IEEE J. Sel. Areas Commun. 23 (10) (2005) 2069–2084, http://dx.doi.org/10.1109/JSAC.2005.854119.

[3] H. Hu, G. Ahn, K. Kulkarni, Detecting and resolving firewall policy anomalies, IEEE Trans. Dependable Secur. Comput. 9 (3) (2012) 318–331, http://dx.doi.org/10.1109/TDSC.2012.20.

[4] S. Pisharody, J. Natarajan, A. Chowdhary, A. Alshalan, D. Huang, Brew: A security policy analysis framework for distributed SDN-based cloud environments, IEEE Trans. Dependable Secur. Comput. 16 (6) (2019) 1011–1025, http://dx.doi.org/10.1109/TDSC.2017.2726066.

[5] D. Bringhenti, S. Bussa, R. Sisto, F. Valenza, Atomizing firewall policies for anomaly analysis and resolution, IEEE Trans. Dependable Secur. Comput. 22 (3) (2025) 2308–2325, http://dx.doi.org/10.1109/TDSC.2024.3495230.

[6] D. Ranathunga, M. Roughan, P. Kernick, N. Falkner, The mathematical foundations for mapping policies to network devices, in: Proc. of the 13th International Joint Conference on E-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT, Lisbon, Portugal, July 26-28, 2016, 2016, pp. 197–206, http://dx.doi.org/10.5220/0005946201970206.

[7] A. El-Hassany, P. Tsankov, L. Vanbever, M.T. Vechev, NetComplete: Practical network-wide configuration synthesis with autocompletion, in: S. Banerjee, S. Seshan (Eds.), Proc. of the 15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018, 2018, pp. 579–594, URL https://www.usenix.org/conference/nsdi18/presentation/el-hassany.

[8] D. Bringhenti, G. Marchetto, R. Sisto, F. Valenza, J. Yusupov, Automated firewall configuration in virtual networks, IEEE Trans. Dependable Secur. Comput. 20 (2) (2023) 1559–1576, http://dx.doi.org/10.1109/TDSC.2022.3160293.

[9] H.H. Hamed, 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, ASIACCS 2006, Taipei, Taiwan, March 21-24, 2006, ACM, 2006, pp. 332–342, http://dx.doi.org/10.1145/1128817.1128867.

[10] E.W. Fulp, Optimization of network firewall policies using directed acyclical graphs, in: Proceedings of the IEEE Internet Management Conference, 2005.

[11] E.M. El-Alfy, S.Z. Selim, On optimal firewall rule ordering, in: 2007 IEEE/ACS International Conference on Computer Systems and Applications (AICCSA 2007), 13-16 May 2007, Amman, Jordan, IEEE Computer Society, 2007, pp. 819–824, http://dx.doi.org/10.1109/AICCSA.2007.370727.

[12] A. Tapdiya, E.W. Fulp, Towards optimal firewall rule ordering utilizing directed acyclical graphs, in: Proceedings of the 18th International Conference on Computer Communications and Networks, IEEE ICCCN 2009, San Francisco, California, USA, August 3-6, 2009, IEEE Computer Society, 2009, pp. 1–6, http://dx.doi.org/10.1109/ICCCN.2009.5235232.

[13] I. Mothersole, M.J. Reed, Optimising rule order for a packet filtering firewall, in: 2011 Conference on Network and Information Systems Security, IEEE, 2001, pp. 1–6, http://dx.doi.org/10.1109/SAR-SSI.2011.5931399.

[14] T. Fuchino, T. Harada, K. Tanaka, K. Mikawa, Acceleration of packet classification using adjacency list of rules, in: 28th International Conference on Computer Communication and Networks, ICCCN 2019, Valencia, Spain, July 29 - August 1, 2019, IEEE, 2019, pp. 1–9, http://dx.doi.org/10.1109/ICCCN.2019.8846923.

[15] T. Fuchino, T. Harada, K. Tanaka, Accelerating packet classification via direct dependent rules, in: 12th International Conference on Network of the Future, NoF 2021, Coimbra, Portugal, October 6-8, 2021, IEEE, 2021, pp. 1–8, http://dx.doi.org/10.1109/NOF52522.2021.9609820.

[16] T. Harada, K. Tanaka, K. Mikawa, A heuristic algorithm for relaxed optimal rule ordering problem, in: P. Hebrard, S. Ghernaouti (Eds.), 2nd Cyber Security in Networking Conference, CSNet 2018, Paris, France, October 24-26, 2018, IEEE, 2018, pp. 1–8, http://dx.doi.org/10.1109/CSNET.2018.8602810.

[17] T. Harada, K. Tanaka, K. Mikawa, Simulated annealing method for relaxed optimal rule ordering, IEICE Trans. Inf. Syst. 103-D (3) (2020) 509–515, http://dx.doi.org/10.1587/TRANSINF.2019FCP0006, URL http://search.ieice.org/bin/summary.php?id=e103-d_3_509.

[18] T. Harada, K. Tanaka, R. Ogasawara, K. Mikawa, A rule reordering method via pairing dependent rules, in: 8th IEEE Conference on Communications and Network Security, CNS 2020, Avignon, France, June 29 - July 1, 2020, IEEE, 2020, pp. 1–9, http://dx.doi.org/10.1109/CNS48642.2020.9162170.

[19] A. Coscia, V. Dentamaro, S. Galantucci, D. Impedovo, A. Maci, A novel genetic algorithm approach for firewall policy optimization, in: Proceedings of the Italian Conference on Cybersecurity (ITASEC 2022), Rome, Italy, June 20-23, 2022, in: CEUR Workshop Proceedings, 3260, CEUR-WS.org, 2022, pp. 236–248, URL https://ceur-ws.org/Vol-3260/paper17.pdf.

[20] A. Coscia, V. Dentamaro, S. Galantucci, A. Maci, G. Pirlo, An innovative two-stage algorithm to optimize firewall rule ordering, Comput. Secur. 134 (2023) 103423, http://dx.doi.org/10.1016/J.COSE.2023.103423.

[21] A. Coscia, A. Maci, N. Tamma, FROG: a firewall rule order generator for faster packet filtering, Comput. Networks 257 (2025) 110962, http://dx.doi.org/10.1016/J.COMNET.2024.110962.

[22] P.N.M. Hansteen, The Book of PF: A No-Nonsense Guide to the OpenBSD Firewall, third ed., No Starch Press, 2014.

[23] Z. Trabelsi, L. Zhang, S. Zeidan, Dynamic rule and rule-field optimisation for improving firewall performance and security, IET Inf. Secur. 8 (4) (2014) 250–257, http://dx.doi.org/10.1049/IET-IFS.2011.0146.

[24] R. Mohan, A. Yazidi, B. Feng, B.J. Oommen, Dynamic ordering of firewall rules using a novel swapping window-based paradigm, in: Proceedings of the 6th International Conference on Communication and Network Security, ICCNS '16, Singapore, November 26-29, 2016, ACM, 2016, pp. 11–20, http://dx.doi.org/10.1145/3017971.3017975.

[25] R. Mohan, A. Yazidi, B. Feng, B.J. Oommen, On optimizing firewall performance in dynamic networks by invoking a novel swapping window-based paradigm, Int. J. Commun. Syst. 31 (15) (2018) http://dx.doi.org/10.1002/DAC.3773.

[26] U. Mustafa, M.M. Masud, Z. Trabelsi, T. Wood, Z.A. Harthi, Firewall performance optimization using data mining techniques, in: 2013 9th International Wireless Communications and Mobile Computing Conference, IWCMC 2013, Sardinia, Italy, July 1-5, 2013, IEEE, 2013, pp. 934–940, http://dx.doi.org/10.1109/IWCMC.2013.6583682.

[27] MAWI Working Group, MAWI working group traffic archive, 2026, https://mawi.wide.ad.jp/mawi/. (Accessed: 03 August 2026).

[28] NETRESEC, SCADA / ICS PCAP files from 4SICS, 2026, https://www.netresec.com/?page=PCAP4SICS. (Accessed: 03 August 2026).

[29] D.E. Taylor, J.S. Turner, ClassBench: a packet classification benchmark, IEEE/ACM Trans. Netw. 15 (3) (2007) 499–511, http://dx.doi.org/10.1145/1295237.1295239, URL http://doi.acm.org/10.1145/1295237.1295239.

[30] J. Matousek, A. Lucanský, D. Janecek, J. Sabo, J. Korenek, G. Antichi, ClassBench-ng: Benchmarking packet classification algorithms in the OpenFlow era, IEEE/ACM Trans. Netw. 30 (5) (2022) 1912–1925, http://dx.doi.org/10.1109/TNET.2022.3155708.

[31] ClassBench-ng Project, ClassBench-ng: Synthetic classification rule sets generator, 2026, https://classbench-ng.github.io/. (Accessed: 03 August 2026).

[32] Applied Research Laboratory, Washington University in St. Louis, ClassBench: A packet classification benchmark, 2026, https://www.arl.wustl.edu/classbench/. (Accessed: 03 August 2026).