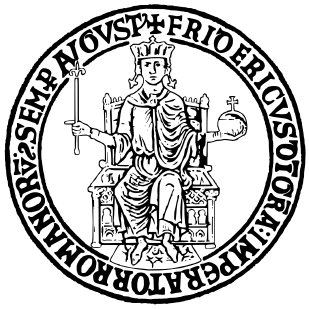
Università degli Studi di Napoli Federico II



Scuola Politecnica e delle Scienze di Base

Dipartimento di Ingegneria Elettrica e Tecnologie dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Network and Cloud Infrastructures

DoS/DDoS Detection SDN Controller

Candidato
Giuseppe Maglione
M63001777

Anno Accademico 2024-2025

Contents

Chapter 1

Introduction

The project builds upon last year's work on detecting and mitigating DoS attacks in a Software Defined Networking (SDN) environment using Mininet and OpenFlow switches. The goal was not to start from scratch, but rather to take last year's implementation as a starting point and significantly improve it by addressing known issues and limitations.

In particular, Chapter [2] describes solutions to 5 critical flaws among the proposed, while Chapter [3] includes some additional features added to enhance the controller logic.

1.1 Network Setup

The topology consists of 3 hosts (*h1*, *h2*, and *h3*) and 4 switches (*s1*, *s2*, *s3*, and *s4*). Each of the four switches (*s1*, *s2*, *s3*, *s4*) is an instance of OVSKernelSwitch, which is the default switch type in Mininet and based on Open vSwitch (OVS). The entire setup is controlled by a remote OpenFlow Ryu controller. The topology file, which remained unchanged from last year's project, establishes the links between hosts and switches, as well as between the switches themselves.

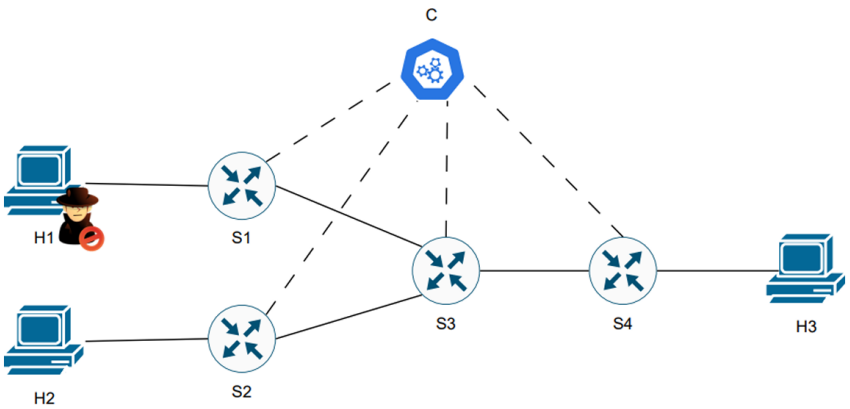


Figure 1.1: Network topology.

Chapter 2

Critical Flaws

The following chapter discusses the strategies used to address five of the critical flaws highlighted. To make the changes made to the project as clear as possible, some sections contains pseudocode or omit portions of code that have not been modified. The complete controller code is available in section [5.1] of Chapter 5.

2.1 Static Threshold

The last year project version uses a static threshold defined based on known topology. In real-world scenarios, these values are unknown or change dynamically, making the method unreliable.

To address this limitation, an adaptive threshold has been introduced. Instead of using a fixed value, the threshold is computed from recent traffic observations. For each switch port, the controller maintains a fixed-length list of the last *N_HISTORY* traffic rates (the *rate_history* structure). This list behaves as a circular buffer: when new values are added, the oldest samples are discarded.

```
# === PARAMETRI GLOBALI ===
timeInterval = 10           # intervallo polling stats
X = 3                       # numero di cicli consecutivi sopra soglia per bloccare
STATIC_THRESHOLD = 7e5      # soglia statica, circa 80% della capacità critica dei link
K = 1                       # fattore di deviazione standard per la soglia dinamica
N_HISTORY = 20              # numero di campioni per valutare soglia dinamica
```

Figure 2.1: Definition of global parameters.

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
        ...
        self.rate_history = {}        # {dpid: {port: [rate1, rate2,...]}}
        ...
        self.monitor_thread = hub.spawn(self._monitor)
```

Figure 2.2: Definition of rate history queue.

The structure update takes place inside the *_port_stats_reply_handler* function, which in the initial project already took care of calculating the rates.

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    ...
    rate_rx = (stat.rx_bytes - previous[stat.port_no][1]) / self.time
    rate_tx = (stat.tx_bytes - previous[stat.port_no][4]) / self.time
    rate = max(rate_rx, rate_tx)  # prendi il rate più alto come indicatore del carico
    ...
    if ev.msg.datapath.id not in self.history:
        self.history[ev.msg.datapath.id] = {}
    if stat.port_no not in self.history[ev.msg.datapath.id]:
        self.history[ev.msg.datapath.id][stat.port_no] = deque(maxlen=N_HISTORY)
    self.history[ev.msg.datapath.id][stat.port_no].append(rate)
```

Figure 2.3: Update rate history.

At each monitoring interval, the controller computes the dynamic threshold as a combination of the mean traffic rate and its standard deviation. This allows the system to adapt to the typical load of each port.

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    ...
    hist = self.history[ev.msg.datapath.id][stat.port_no]
    if len(hist) >= 2:
        mean = np.mean(hist)
        stdev = np.std(hist)
        dynamic_threshold = mean + K * stdev
    else:
        dynamic_threshold = STATIC_THRESHOLD
```

Figure 2.4: Evaluation of dynamic threshold.

Once the necessary data structure were defined and the dynamic threshold was calculated, it is simply necessary to change the current rate check.

```
# se per X cicli la threshold è superata, allora blocca.
threshold = max(dynamic_threshold, STATIC_THRESHOLD)
if  rate > threshold:
    # incrementa il contatore per la porta
    ...
elif rate <= threshold:
    # decrementa il contatore per la porta
    ...
```

Figure 2.5: Check if rate exceeds the threshold.

It's important to note that although a dynamic threshold has been introduced, the static threshold value is still used. This is because considering only the dynamic threshold could lead to situations where the controller raises alarms also for hosts producing normal traffic. To better understand why, let's consider the following example: Suppose that at a given instant the network is stationary, so the calculated mean and standard deviation will be 0, and consequently the dynamic threshold will also be 0. Let's then assume that a host suddenly starts producing normal traffic. Since the dynamic threshold is based on the average of previous values, the host's rate will surely exceed the threshold, and therefore the sudden presence of traffic is interpreted by the controller as an alarm.

To avoid this problem, the static threshold is still used as a lower bound: the final threshold is defined as the maximum between the static and the dynamic one. This ensures robustness to both sudden traffic changes and long-term variations.

2.2 Controller-Centric Blocking Decisions

In the current project version, only the controller makes blocking decisions, based on its monitoring logic. This prevents external apps or admins from contributing to policies, limiting extensibility.

To solve this limitation, a shared data structure (*policies*) has been introduced to store all currently active policies. This way, any external module can contribute to policy management by simply interacting with the structure.

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
        ...
        self.rate_history = {}        # {dpid: {port: [rate1, rate2,...]}}
        self.policies = {}            # {dpid: {port: {"blocked": bool, "reason": str, "timestamp": float}}}
        ...
        self.monitor_thread = hub.spawn(self._monitor)
```

Figure 2.6: Definition of policies data structure.

Each policy is defined by a set of attributes:

- *blocked*: Boolean value indicating the validity of the policy.
- *reason*: Reason for the application.
- *timestamp*: When the policy has been applied.

With the introduction of this data structure, the monitoring loop no longer needs to apply the block directly, but simply update the *policies* in the *_port_stats_reply_handler* function. The *enforce_policy* function will then monitors the *policies* structure for changes and applies or removes the corresponding OpenFlow rules by calling the *lock_flow* or *unlock_flow* functions, which remained unchanged from last year's implementation.

```
...
# X cicli sopra la soglia → blocco della porta
if  self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] == X:
    print(RED + "ALLARME SULLA PORTA " + str(stat.port_no) + " dello Switch " + str(ev.msg.datapath.id) + RESET)
    self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][1] =  1
    print(self.alarm_switch_port)
    time.sleep(1)
    self.policies[dpid][port_no]["blocked"] = True

# contatore raggiunge 0 → sblocco della porta
elif self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][0] == 0:
    self.alarm_switch_port[ev.msg.datapath.id][stat.port_no][1] = 0
    self.policies[dpid][port_no]["blocked"] = False

# enforcement dopo update policy
self.enforce_policies()
...
```

Figure 2.7: Update policy if a block occurs.

Note: The implementation of the polocies enforcement function is omitted here, as it is directly reported in the next section [2.13].

The next step is to enable administrators to actively contribute to policy management. For this purpose, a REST module has been implemented directly within the controller. Since Ryu already provides a lightweight integrated WSGI server, it was sufficient to define RESTful APIs that expose policy management functions (addition, removal, and listing).

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
        ...
        # setup REST server
        wsgi = kwargs['wsgi']
        wsgi.register(PolicyRESTController, {'app': self})
        ...
        self.monitor_thread = hub.spawn(self._monitor)
```

Figure 2.8: Initialize REST server.

Finally, the class that exposes the RESTful API has been implemented.

```
# === WEB APP CONTROLLER ===
class PolicyRESTController(ControllerBase):

    def __init__(self, req, link, data, **config):
        super(PolicyRESTController, self).__init__(req, link, data, **config)
        self.app = data['app']

    @route('policy', '/policy', methods=['GET'])
    def list_policies(self, req, **kwargs):
        return Response(content_type='application/json',
                        body=json.dumps(self.app.policies))

    @route('policy', '/policy', methods=['POST'])
    def add_policy(self, req, **kwargs):
        try:
            policy = req.json if req.body else {}
            dpid = int(policy['switch'])
            port = int(policy['port'])
            reason = policy.get('reason', 'manual')

            if dpid not in self.app.policies:
                self.app.policies[dpid] = {}
            self.app.policies[dpid][port] = {"blocked": True, "reason": reason}

            # applica la policy
            dp = self.app.datapaths.get(dpid)
            if dp:
                self.app.enforce_policiest(dpid)
            return Response(status=200, body="Policy added")

        except Exception as e:
            return Response(status=400, body=str(e))

    @route('policy', '/policy', methods=['DELETE'])
    def delete_policy(self, req, **kwargs):
        try:
            policy = req.json if req.body else {}
            dpid = int(policy['switch'])
            port = int(policy['port'])

            if dpid in self.app.policies and port in self.app.policies[dpid]:
                self.app.policies[dpid][port]['blocked'] = False

                # rimuovi la policy
                dp = self.app.datapaths.get(dpid)
                if dp:
                   self.app.enforce_policiest(dpid)
            return Response(status=200, body="Policy removed")

        except Exception as e:
            return Response(status=400, body=str(e))
```

Figure 2.9: Implementation of RESTful API class.

This way, any network administrator can interact with the *policies* data structure using HTTP methods (*GET*, *POST*, *DELETE*). For example, using the command line tool *curl*, it is possible to:

- View current policies.

```
curl -X GET http://127.0.0.1:8080/policy
```

- Manually block a switch port.

```
curl -X POST -H "Content-Type: application/json" \
 -d '{"switch":1, "port":1, "reason":"admin"}' \
 http://127.0.0.1:8080/policy
```

- Manually unlock a switch port.

```
curl -X DELETE -H "Content-Type: application/json" \
     -d '{"switch":1, "port":1}' \
     http://127.0.0.1:8080/policy
```

In section [3.2], a web-app - that allows to perform everything stated above via graphical interface - has been developed to facilitate policy management.

2.2.1 Possible Vulnerability

Currently, no authentication is enforced on the REST API. This means that anyone with access to the controller's REST endpoint could add or remove policies by simply writing the right HTTP query. As a future improvement, an authentication and session management mechanism should be introduced to restrict access to authorized administrators only. For example, using a session cookie provided to logged users only.

2.3 Lack of Modular Detection and Mitigation Design

Actually, monitoring, decision-making, and enforcement are not clearly separated. Everything is done within the same function. This structure is hard to maintain or extend, because replacing one part affects the whole system.

To address this issue, the system has been refactored to separate monitoring, detection, and enforcement into distinct threads:

- *Monitoring*: Periodically polls switches for port statistics and updates shared *stats_queue* structures.
- *Detection*: Consumes the monitoring data, applies decision logic, and if necessary inserts blocking policies into the shared *policy_queue* structure.
- *Enforcement*: Periodically checks the *policy_queue* data structure and applies or removes OpenFlow rules based on new or updated policies.

The main problem with this new division is the possibility of concurrency between different threads. To safely exchange information between threads, thread-safe queues from Python's *queue* module are used:

- *stats_queue*: Shared between the monitoring and detection threads, containing port statistics.
- *policy_queue*: Shared between the detection and enforcement threads, containing policies to be applied.

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.mac_to_port = {}

        # strutture dati
        self.monitoring_stats = {}
        self.detection_state = {}     # {dpid: {port: {"blocked": False, "counter": 0}}}
        self.rate_history = {}        # {dpid: {port: [rate1, rate2,...]}}
        self.policies = {}            # {dpid: {port: policy}}

        # setup REST server
        wsgi = kwargs['wsgi']
        wsgi.register(PolicyRESTController, {'app': self})

        # code thread-safe per comunicazione tra thread
        self.stats_queue = queue.Queue()     # monitoring → detection
        self.policy_queue = queue.Queue()    # detection → enforcement

        # spawn dei thread
        self.monitor_thread = hub.spawn(self._monitor_loop)
        self.detect_thread = hub.spawn(self._detect_loop)
        self.enforce_thread = hub.spawn(self._enforce_loop)
```

Figure 2.10: Definition of thread-safe data structures.

Before moving on to the code, it is a good idea to outline the current controller structure and how threads communicate with each other using a flowchart.

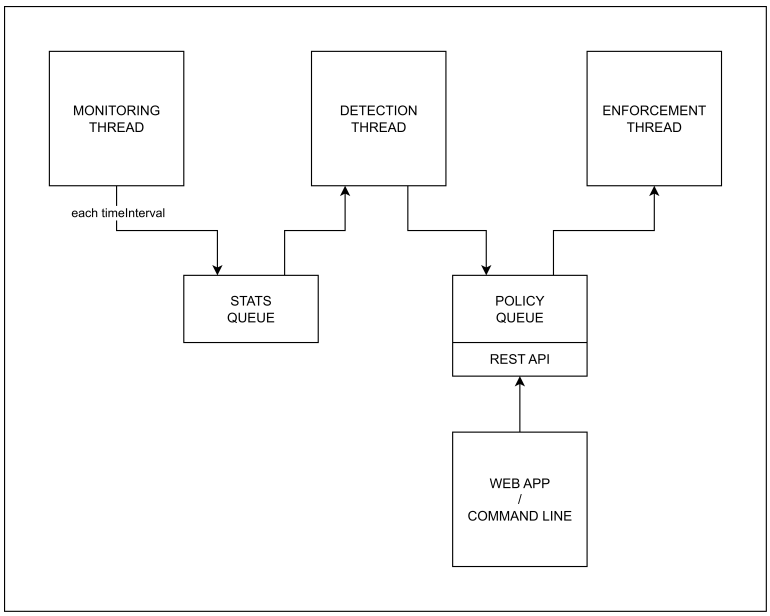


Figure 2.11: Flowchart of structure and threads communication.

Now let's move on to the implementations of the functions executed by each thread.

```
# === MONITORING ===
@set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    dp = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        self.datapaths[dp.id] = dp
    elif ev.state == DEAD_DISPATCHER:
        self.datapaths.pop(dp.id, None)

def _request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
    datapath.send_msg(req)

def _monitor_loop(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
        hub.sleep(timeInterval)
        self.logger.info(BLUE + '\t[MONITORING] New stats produced' + RESET)
```

```
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    dpid = ev.msg.datapath.id
    body = ev.msg.body

    previous = self.monitoring_stats.get(dpid, {})
    current = {}

    for stat in sorted(body, key=attrgetter('port_no')):
        current[stat.port_no] = [stat.rx_packets, stat.rx_bytes,
                                 stat.rx_errors, stat.tx_packets,
                                 stat.tx_bytes, stat.tx_errors]

        if stat.port_no in previous:
            rate_rx = (stat.rx_bytes - previous[stat.port_no][1]) / timeInterval
            rate_tx = (stat.tx_bytes - previous[stat.port_no][4]) / timeInterval
            rate = max(rate_rx, rate_tx)

            # usa la storia dei rate per calcolare la soglia dinamica
            self.rate_history.setdefault(dpid, {}).setdefault(stat.port_no, []).append(rate)
            hist = self.rate_history[dpid][stat.port_no]
            if len(hist) > N_HISTORY:
                hist.pop(0)

            if hist:
                mean_val = np.mean(hist)
                std_val = np.std(hist)
                dynamic_threshold = mean_val + K * std_val
            else:
                dynamic_threshold = STATIC_THRESHOLD

            # push in coda condivisa
            self.stats_queue.put((dpid, stat.port_no, rate, dynamic_threshold))

    # aggiorna le statistiche precedenti con le statistiche correnti per il prossimo ciclo
    self.monitoring_stats[dpid] = current
```

Figure 2.12: Monitoring thread functions.

```
# === DETECTION ===
def _detect_loop(self):
    while True:
        try:
            dpid, port, rate, dynamic_threshold = self.stats_queue.get(timeout=timeInterval)
        except queue.Empty:
            continue

        # inizializza lo stato se non esiste
        self.detection_state.setdefault(dpid, {}).setdefault(port, {
            "blocked": False,
            "counter": 0
        })
        state = self.detection_state[dpid][port]

        threshold = max(dynamic_threshold, STATIC_THRESHOLD)

        if rate > threshold:
            self.logger.info(YELLOW + f"\t[DETECTION] Warning: A possible DoS attack has been detected. Info:" +RESET)
            self.logger.info(YELLOW + f"Switch: {dpid}, Port: {port}, Exceeded threshold: {threshold:.1f},
                                      Current rate: {rate:.1f}, Counter: {statre['counter']}/{X}" + RESET)
            # incrementa contatore se il rate supera la soglia
            state["counter"] += 1
        else:
            # decrementa contatore se il rate è sotto la soglia
            state["counter"] -= 1

        # porta non bloccata → valuta blocco
        if not state["blocked"] and state["counter"] >= X:
            policy = {
                "switch": dpid,
                "port": port,
                "blocked": True,
                "reason": "DoS detection"
            }
            # push policy in coda condivisa
            self.policy_queue.put(policy)
            state["blocked"] = True
            state["counter"] = 0

        # porta bloccata → valuta sblocco
        elif state["blocked"] and state["counter"] == 0:
            # push richiesta sblocco in coda condivisa
            self.policy_queue.put({
                "action": "remove",
                "switch": dpid,
                "port": port
            })
            state["blocked"] = False
            state["counter"] = 0
```

Figure 2.13: Detection thread function.

```python
# === ENFORCEMENT ===
def _enforce_loop(self):
    while True:
        try:
            policy = self.policy_queue.get(timeout=1)
        except queue.Empty:
            continue

        dpid = str(policy.get("switch"))
        port = str(policy.get("port"))
        reason = str(policy.get("reason", "manual"))

        # se si tratta di una richiesta di sblocco, rimuovi la policy
        if policy.get("action") == "remove":
            if dpid in self.policies and port in self.policies[dpid]:
                del self.policies[dpid][port]
            dp = self.datapaths.get(int(dpid))
            if dp:
                self.unlock_flow(dp, int(port))
            continue

        if "blocked" not in policy:
            policy["blocked"] = True
            policy["reason"] = reason

        self.policies.setdefault(dpid, {})[port] = policy
        dp = self.datapaths.get(int(dpid))
        if dp and policy["blocked"]:
            self.lock_flow(dp, int(port))
```

```python
def lock_flow(self, dp, port_no):
    parser = dp.ofproto_parser
    ofproto = dp.ofproto
    match = parser.OFPMatch(in_port=port_no)
    mod = parser.OFPFlowMod(datapath=dp, priority=2, match=match, instructions=[],
                            command=ofproto.OFPFC_ADD, out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
    dp.send_msg(mod)
    print(RED + f"\t[ENFORCEMENT] Blocked traffic on port {port_no} of switch {dp.id}" + RESET)

def unlock_flow(self, dp, port_no):
    parser = dp.ofproto_parser
    ofproto = dp.ofproto
    match = parser.OFPMatch(in_port=port_no)
    mod = parser.OFPFlowMod(datapath=dp, priority=2, match=match,
                            command=ofproto.OFPFC_DELETE, out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
    dp.send_msg(mod)
    print(GREEN + f"\t[ENFORCEMENT] Unblocked traffic on port {port_no} of switch {dp.id}" + RESET)
```

Figure 2.14: Enforcement thread functions.

It's worth noting that after the refactoring a slightly different policy removal logic has adopted. Now, a particular policy is inserted into the *policies* structure with an action field set to the value "*remove*."

2.4 Over Blocking

The mitigation strategy blocks all traffic on a switch port once a throughput threshold is exceeded. Also it may block it for more switches than needed. The problem is that also legitimate traffic is dropped.

To overcome this limitation, a finer-grained monitoring and blocking logic was introduced. The controller has been refined to operate on a per-host basis using the source MAC address. Consequently, the previously defined data structures no longer associate statistics solely with a port, but track the traffic generated by each host.

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
    self.datapaths = {}
    self.mac_to_port = {}

    # strutture dati
    self.port_stats = {}          # {dpid: {port: {"rx_bytes": ..., "rx_packets": ...}}}
    self.prev_port_stats = {}     # stats al ciclo precedente

    self.detection_state = {}     # {dpid: {mac: {"blocked": False, "counter": 0}}}
    self.rate_history = {}        # {dpid: {mac: [rate1, rate2,...]}}
    self.policies = {}            # {dpid: {mac: policy}}

    # setup REST per web-app
    wsgi = kwargs['wsgi']
    wsgi.register(PolicyAPI, {'controller': self})

    # code thread-safe per comunicazione tra thread
    self.stats_queue = queue.Queue()     # monitoring → detection
    self.policy_queue = queue.Queue()    # detection → enforcement

    # spawn dei thread
    self.monitor_thread = hub.spawn(self._monitor_loop)
    self.detect_thread = hub.spawn(self._detect_loop)
    self.enforce_thread = hub.spawn(self._enforce_loop)
```

Figure 2.15: Definition of updated data structures.

It is important to note that while rate calculations are now MAC-specific, the underlying OpenFlow statistics remain collected at the port level. The *mac_to_port* mapping enables the controller to attribute port-level traffic to the correct source host.

The core logic of monitoring, detection and enforcement remains completely unchanged. Since only minor modifications were made to the functions of these threads, their implementation is omitted here.
Below is only reported the code for the *_port_stats_reply_handler* function, which was modified to to save statistics per MAC address rather than per port.

```python
@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    dp = ev.msg.datapath
    dpid = dp.id
    self.port_stats.setdefault(dpid, {})
    self.prev_port_stats.setdefault(dpid, {})

    # aggiorna le statistiche correnti con i dati del reply
    for stat in ev.msg.body:
        port_no = stat.port_no
        self.port_stats[dpid][port_no] = stat

    # calcola il rate di traffico per ogni porta
    for port_no, stat in self.port_stats[dpid].items():
        if port_no in self.prev_port_stats[dpid]:
            prev_stat = self.prev_port_stats[dpid][port_no]
            # calcola la differenza in byte e il tempo trascorso
            delta_bytes = stat.rx_bytes - prev_stat.rx_bytes
            delta_time = timeInterval

            # calcola il rate in byte/s
            rate = (delta_bytes / delta_time) if delta_time > 0 else 0

            # trova il MAC address associato alla porta corrente
            mac_to_check = None
            for mac, port in self.mac_to_port.get(dpid, {}).items():
                if port == port_no and mac in HOST_MACS:
                    mac_to_check = mac
                    break

            if mac_to_check:
                # usa la storia dei rate per calcolare la soglia dinamica
                self.rate_history.setdefault(dpid, {}).setdefault(mac_to_check, []).append(rate)
                hist = self.rate_history[dpid][mac_to_check]
                if len(hist) > N_HISTORY:
                    hist.pop(0)

                if hist:
                    mean_val = np.mean(hist)
                    std_val = np.std(hist)
                    dynamic_threshold = mean_val * MEAN_RATIO + K * std_val
                else:
                    dynamic_threshold = STATIC_THRESHOLD

                # push delle stats in coda detection
                self.stats_queue.put((dpid, mac_to_check, rate, dynamic_threshold))

    # aggiorna le statistiche precedenti con le statistiche correnti per il prossimo ciclo
    self.prev_port_stats[dpid] = self.port_stats[dpid].copy()
```

Figure 2.16: Updated monitoring function.

It is worth noting that the introduced modification requires a change in the rate calculation logic, now diversified by host and no longer by port. In particular, for each switch port, the rate is computed as the difference in bytes between two consecutive polling intervals, divided by the elapsed time:

$$rate = \frac{rx - bytes_{current} - rx - bytes_{previous}}{timeInterval}$$

This provides the average incoming traffic rate (in bytes per second) for the given port during the last interval. Once the traffic rate for a port is computed, the controller checks *mac_to_port* structure to identify which host MAC is connected

to that port. Only hosts that belong to the predefined set *HOST_MACS* are considered for detection, to exclude switch-to-switch traffic.

2.4.1 Possible Vulnerability

Performing MAC-based traffic control inevitably introduces a vulnerability: MAC spoofing. An attacker could write a simple script that launches a DoS attack and change it's MAC address after being blocked. This can be achieved with tools like *macchanger*, which allows to change the MAC address of the attacker machine with a random one. This way, the generated blocking policy is completely ineffective and the attacker can continue undisturbed.

2.5 Detection Limited to Classic DoS Patterns

Only high-throughput, constant-rate DoS attacks are detected. The detection and mitigation strategy is tailored just for one attacker. Fails against stealthy attacks (not constant bitrate) or DDoS with distributed, bursty traffic.

To overcome these limitations, a Machine Learning (ML) model has been integrated into the controller.

2.5.1 Offline Phase (Training)

In order to introduce the ML model into the controller, it was first necessary to define the features on which it had to work and generate instances to train it.

- *Statistical properties of traffic rates (mean, variance, min and max rate)*: Describe the average and variability of traffic rates.
- *Inter-packet intervals (mean and standard deviation)*: Reveal burstiness or irregular sending patterns.
- *Burst count*: Counts how often the rate exceeds a multiple of its mean, detecting sudden traffic spikes.
- *Source diversity per destination*: Measures how many different hosts contact the same target, useful to detect DDoS from multiple sources.
- *Small-size packet ratio*: Indicates if many small packets are sent, often indicative of stealthy DoS attacks.

The statistics from which the values of these features are calculated are stored in new data structures, populated inside the *_packet_in_handler* function.

```
def __init__(self, *args, **kwargs):
    super(SimpleSwitch13, self).__init__(*args, **kwargs)
    ...
    # setup per raccolta istanze (ML)
    self.mac_timestamps = {}      # {dpid: {mac: [t1, t2, ...]}}
    self.dst_sources = {}         # {dpid: {dst: set([src1, src2, ...])}}
    self.mac_pkt_sizes = {}       # {dpid: {mac: [pkt_len1, pkt_len2, ...]}}
    self.csv_file = "traffic_features.csv"
    self.csv_fields = [
        "timestamp","dpid","mac",
        "mean_rate","var_rate","max_rate","min_rate",
        "mean_interval","std_interval","burst_count",
        "src_diversity","small_pkt_ratio","class"
    ]
    # creazione del file csv
    if not os.path.exists(self.csv_file) or os.path.getsize(self.csv_file) == 0:
        with open(self.csv_file, "w", newline="") as f:
            writer = csv.DictWriter(f, fieldnames=self.csv_fields)
            writer.writeheader()
    # setup modello (RandomForest)
    self.ml_model = joblib.load("dos_ddos_detector.pkl")
    self.ml_model.n_jobs = 1
    self.ml_features = [
        "mean_rate","var_rate","max_rate","min_rate",
        "mean_interval","std_interval","burst_count",
        "src_diversity","small_pkt_ratio"
    ]
    ...
    # thread di raccolta dati per addestrare il modello
    self.feature_thread = hub.spawn(self._feature_loop)
```

Figure 2.17: Definition of new data structures and setup ML model.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    ...
    # --- raccolta dati per modello ML --- (unica parte di codice non fornito)
    now = time.time()

    # aggiorna mappa dei sorgenti che contattano una stessa destinazione
    self.dst_sources.setdefault(dpid, {}).setdefault(dst, set()).add(src)
    # tieni solo le ultime N_HISTORY destinazioni
    if len(self.dst_sources[dpid][dst]) > N_HISTORY:
        self.dst_sources[dpid][dst].pop()

    # aggiorna lista dimensioni pacchetti per ogni sorgente
    pkt_len = len(msg.data)
    self.mac_pkt_sizes.setdefault(dpid, {}).setdefault(src, []).append(pkt_len)
    # tieni solo gli ultimi N_HISTORY pacchetti
    if len(self.mac_pkt_sizes[dpid][src]) > N_HISTORY:
        self.mac_pkt_sizes[dpid][src].pop(0)

    # aggiorna timestamp pacchetti per intervalli
    self.mac_timestamps.setdefault(dpid, {}).setdefault(src, []).append(now)
    # tieni solo gli ultimi N_HISTORY timestamp
    if len(self.mac_timestamps[dpid][src]) > N_HISTORY:
        self.mac_timestamps[dpid][src].pop(0)
    # --- fine raccolta dati ---
    ...
```

Figure 2.18: Update feature data structures for ML model.

To transform the acquired raw statistics into instances usable by the model, a thread has been added that periodically calculates the values and writes them

to a *csv* file.

```
# === MACHINE LEARNING MODEL ===
def extract_features(self, dpid, mac):
    rates = self.rate_history.get(dpid, {}).get(mac, [])
    times = self.mac_timestamps.get(dpid, {}).get(mac, [])

    if len(rates) < 1 or len(times) < 1:
        return None  # non abbastanza dati

    mean_rate = np.mean(rates)
    var_rate = np.var(rates)
    max_rate = np.max(rates)
    min_rate = np.min(rates)

    intervals = np.diff(times)
    mean_interval = np.mean(intervals) if len(intervals) > 0 else 0
    std_interval = np.std(intervals) if len(intervals) > 0 else 0

    # burst count: quante volte il rate supera 1.5x la media
    burst_count = sum(r > mean_rate * 1.5 for r in rates)

    # numero di sorgenti per la stessa destinazione,
    # prende la media dei sorgenti unici visti
    # nelle destinazioni contattate da questo MAC.
    # (indicativo di possibili attacchi distribuiti)
    dst_map = self.dst_sources.get(dpid, {})
    src_diversity = 0
    if dst_map:
        counts = [len(srcs) for srcs in dst_map.values()]
        if counts:
            src_diversity = sum(counts) / len(counts)

    # percentuale pacchetti piccoli (<250B)
    # (indicativo di possibili attacchi stealth)
    pkt_sizes = self.mac_pkt_sizes.get(dpid, {}).get(mac, [])
    small_pkt_ratio = 0
    if pkt_sizes:
        small_pkt_ratio = sum(1 for s in pkt_sizes if s < 250) / len(pkt_sizes)

    features = {
        "timestamp": time.time(),
        "dpid": dpid,
        "mac": mac,
        "mean_rate": mean_rate,
        "var_rate": var_rate,
        "max_rate": max_rate,
        "min_rate": min_rate,
        "mean_interval": mean_interval,
        "std_interval": std_interval,
        "burst_count": burst_count,
        "src_diversity": src_diversity,
        "small_pkt_ratio": small_pkt_ratio,
        "class": str(FEATURE_CLASS)
    }
    return features
```

Figure 2.19:

```python
def _feature_loop(self):
    while True:
        hub.sleep(timeInterval)  # ogni timeInterval estrai e salva
        for dpid, macs in self.rate_history.items():
            for mac in macs.keys():
                if mac not in HOST_MACS:  # salta MAC non host
                    continue
                features = self.extract_features(dpid, mac)
                if features:
                    features["class"] = "attacco"  # aggiungi colonna classe
                    with open(self.csv_file, "a", newline="") as f:
                        writer = csv.DictWriter(
                            f,
                            fieldnames=self.csv_fields,
                            extrasaction="ignore",
                            restval=""
                        )
                        writer.writerow(features)
        print(CYAN + f"\t[FEATURE EXTRACTOR] New features extracted" + RESET)
```

Figure 2.20: ML model thread functions.

Once the setup for collecting data for the model has been prepared, all that remains is to generate traffic in the network in order to obtain positive instances (normal traffic) and negative instances (traffic under attack). To do this, scripts were prepared to be executed by hosts *h1* and *h2*. In both cases, host *h3* listens on both UDP and TCP ports.

For the generation of traffic classified as normal, the following scripts were used.

```bash
#!/bin/bash

TARGET=10.0.0.3
PORT_UDP=5001
PORT_TCP=5002

echo "[H1] Avvio traffico NORMALE"

for cycle in {1..5}; do
    echo "[H1] Ciclo $cycle in corso..."

    # 1. UDP leggero
    iperf -c $TARGET -p $PORT_UDP -u -b 1M -t 35
    # 2. TCP medio
    iperf -c $TARGET -p $PORT_TCP -b 2M -t 25
    # 3. UDP leggero
    iperf -c $TARGET -p $PORT_UDP -u -b 500K -t 20
    # 4. Ping variabile
    for i in {1..30}; do
        SIZE=$((50 + RANDOM % 50))
        SLEEP=$(awk -v min=0.2 -v max=0.8 'BEGIN{s=min+rand()*(max-min); print s}')
        ping -c 1 -s $SIZE $TARGET > /dev/null
        sleep $SLEEP
    done

    echo "[H1] Ciclo $cycle completato."
    SLEEP=$(awk -v min=0 -v max=1 'BEGIN{s=min+rand()*(max-min)}')
    sleep $SLEEP
done

echo "[H1] Script completato!"
```

Figure 2.21: Normal-traffic script for *h1*.

```bash
#!/bin/bash

TARGET=10.0.0.3
PORT_UDP=5003
PORT_TCP=5004

echo "[H2] Avvio traffico NORMALE"

for cycle in {1..5}; do
    echo "[H2] Ciclo $cycle in corso..."

    # 1. UDP medio
    iperf -c $TARGET -p $PORT_UDP -u -b 2M -t 25
    # 2. TCP leggero
    iperf -c $TARGET -p $PORT_TCP -b 1M -t 35
    # 3. UDP leggero
    iperf -c $TARGET -p $PORT_UDP -u -b 500K -t 20
    # 4. Ping variabile
    for i in {1..30}; do
        SIZE=$((50 + RANDOM % 100))
        SLEEP=$(awk -v min=0.3 -v max=1 'BEGIN{s=min+rand()*(max-min); print s}')
        ping -c 1 -s $SIZE $TARGET > /dev/null
        sleep $SLEEP
    done

    echo "[H2] Ciclo $cycle completato."
    SLEEP=$(awk -v min=1 -v max=3 'BEGIN{s=min+rand()*(max-min)}')
    sleep $SLEEP
done

echo "[H2] Script completato!"
```

Figure 2.22: Normal-traffic script for *h2*.

While to generate the traffic under attack, simulating both stealth and DDoS attacks (*h1* and *h2* attack together), the following scripts were used.

```bash
#!/bin/bash

TARGET=10.0.0.3
PORT_UDP=5001
PORT_TCP=5002

echo "[H1] Avvio traffico di ATTACCO"

for cycle in {1..20}; do
    echo "[H1] Ciclo $cycle in corso..."

    # 1. UDP flood (DDoS con h2)
    iperf -c $TARGET -p $PORT_UDP -u -b 4M -t 25

    # 2. TCP flood (DDoS con h2)
    iperf -c $TARGET -p $PORT_TCP -b 6M -t 25

    echo "[H1] Ciclo $cycle completato."
done

echo "[H1] Script completato!"
```

Figure 2.23: DDoS attack-traffic script for *h1*.

```bash
#!/bin/bash

TARGET=10.0.0.3
PORT_UDP=5003
PORT_TCP=5004

echo "[H2] Avvio traffico di ATTACCO"

for cycle in {1..20}; do
    echo "[H2] Ciclo $cycle in corso..."

    # 1. UDP flood (DDoS con h1)
    iperf -c $TARGET -p $PORT_UDP -u -b 6M -t 25

    # 2. TCP flood (DDoS con h1)
    iperf -c $TARGET -p $PORT_TCP -b 4M -t 25

    echo "[H2] Ciclo $cycle completato."
done

echo "[H2] Script completato!"s
```

Figure 2.24: DDoS attack-traffic script for *h2*.

```bash
#!/bin/bash

TARGET=10.0.0.3

echo "[H1] Avvio attacco DoS Stealth"

while true; do
    SIZE=$((100 + RANDOM % 600))
    SLEEP=$(awk -v min=0.02 -v max=0.12 'BEGIN{s=min+rand()*(max-min); print s}')
    ping -c 1 -s $SIZE $TARGET > /dev/null
    sleep $SLEEP
done

echo "[H1] Attacco completato!"
```

Figure 2.25: Stealth attack-traffic script for *h1*.

For the stealth attack executed by host *h1*, host *h2* simply generated low-rate UDP traffic towards host *h3*.

After collecting enough instances - approximately 1000 for normal traffic and 1000 for traffic under attack - the model was trained.

```python
import pandas as pd
import numpy as np
from sklearn.model_selection import StratifiedKFold, cross_val_score, cross_validate
from sklearn.ensemble import RandomForestClassifier
from sklearn.preprocessing import LabelEncoder

# carica dataset
df = pd.read_csv("traffic.csv")

# colonne da usare come feature
X = df.drop(columns=["timestamp", "dpid", "mac", "class"])
y = LabelEncoder().fit_transform(df["class"])  # 0 = attacco, 1 = normale

# crea modello
clf = RandomForestClassifier(
    n_estimators=200,
    max_depth=10,
    random_state=42,
    n_jobs=-1
)

# cross validation
cv = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

scoring = ["accuracy", "precision", "recall", "f1"]

results = cross_validate(clf, X, y, cv=cv, scoring=scoring, return_train_score=False)

# stampa risultati
print("\n=== Cross Validation (5-fold) ===")
for metric in scoring:
    print(f"{metric.capitalize():<10}: {results[f'test_{metric}'].mean():.4f} "
          f"(± {results[f'test_{metric}'].std():.4f})")

# allena il modello finale su tutto il dataset
clf.fit(X, y)

# stampa importanza delle feature
importances = pd.Series(clf.feature_importances_, index=X.columns)
print("\nFeature Importances:")
print(importances.sort_values(ascending=False))

# salva modello
import joblib
joblib.dump(clf, "dos_ddos_detector.pkl")
print("\n[INFO] Modello salvato")
```

Figure 2.26: Code for training the model.

2.5.2 Online Phase (Integration)

Once the offline training phase was completed, the model was inserted into the controller. During runtime, the detection thread uses the *extract_features* function to obtain traffic data, then used to query the model for predictions. If it classifies a host as potentially malicious, the corresponding alarm counter is incremented.

```
# === DETECTION ===
def _detect_loop(self):
    while True:
        try:
            dpid, mac, rate, dynamic_threshold = self.stats_queue.get(timeout=timeInterval)
        except queue.Empty:
            continue
        ...
        if not state["blocked"]:
            # --- detection classica ---
            classical_detect = rate > threshold
            if rate < threshold:
                state["counter"] = max(0, state["counter"] - 1)     # decrementa contatore
            # --- detection machine learning ---
            ml_detect = False
            if mac in HOST_MACS:    # evita predizioni su dati degli switch (solo host → host)
                features = self.extract_features(dpid, mac)
                if features:
                    # evita predizione su dati incompleti
                    if any(features[f] > 0 for f in self.ml_features):
                        X_input = pd.DataFrame([[features[f] for f in self.ml_features]],
                                               columns=self.ml_features)
                        try:
                            pred = self.ml_model.predict(X_input)
                            if pred[0] == 0:        # attacco = 0, normale = 1
                                ml_detect = True
                        except Exception as e:
                            hub.sleep(1)
            if classical_detect:
                # incrementa il contatore allarme del MAC
                ...
            if ml_detect:
                # incrementa il contatore allarme del MAC
                ...
```

Figure 2.27: Integration of ML model in the detection logic.

2.5.3 Possible Vulnerability

The ML model is trained offline using traffic traces collected on a specific network topology. As a result, it is highly dependent on that topology: any changes, such as added hosts, modified links, or altered traffic patterns, may reduce its accuracy or make it ineffective. A potential improvement could be implementing an online learning, allowing the model to continuously adapt to evolving network conditions and maintain detection effectiveness.

Chapter 3

Additional Features

The following chapter illustrates some of the additional features that have been added to the controller, in order to provide better blocking logic and greater simplicity in policy management for network administrator. Similar to what was reported in the previous chapter, in some figures pseudocode is used and some portions of code have been omitted, to make the changes easier to understand. The complete controller code is available in section [5.1] of Chapter 4.

3.1 Time-based Unblocking Logic

The current unlocking logic is based on an alarm counter, just like the locking logic. This makes the punishments for malicious hosts not severe enough.

To mitigate this problem, the blocking logic has undergone an evolution: instead of using an unblock counter that updates at each controller cycle (every *timeInterval*), blocking occurs in terms of time. Let's try to understand why this change was necessary.

In the initial version of the project, an attacker was blocked if it exceeded the threshold for X consecutive cycles, and then unblocked once the alarm counter reached zero. The counter itself was reduced whenever the observed rate dropped below the threshold. This approach, however, introduced an unintended side effect. Once an attacker was blocked, its traffic was immediately filtered by the switch, meaning its observed rate inevitably dropped close to zero. As a consequence, in the following cycles the counter was decremented and quickly reset to zero, leading the system to automatically unblock the attacker after a short time, even if the malicious behavior had not stopped. To address this issue, the counter-based logic was replaced with a time-based unblocking strategy. Now, once a host is blocked, it is associated with an *unlock_time* proportional to how much the rate exceeds the threshold (according to a constant *BAN_RATIO* factor).

```python
# === DETECTION ===
def _detect_loop(self):
    while True:
        try:
            dpid, mac, rate, dynamic_threshold = self.stats_queue.get(timeout=timeInterval)
        except queue.Empty:
            continue

        now = time.time()

        # gestisci eventuali sblocchi
        for dp_id, hosts in self.detection_state.items():
            for mac_addr, state in hosts.items():
                if state.get("blocked") and now >= state.get("unlock_time", 0):
                    self.policy_queue.put({
                        "action": "remove",
                        "switch": dp_id,
                        "mac": mac_addr
                    })
                    state["blocked"] = False
                    state["counter"] = 0  # reset contatore dopo sblocco
        ...
        if not state["blocked"]:
            # detection classica + detection ML
            ...
            # blocco se il contatore raggiunge X
            if state["counter"] >= X:
                if ml_detect:
                    block_time = BAN_TIME
                else:
                    excess_ratio = rate / threshold
                    block_time = excess_ratio * BAN_RATIO * X
                state["unlock_time"] = now + block_time

                policy = {
                    "switch": dpid,
                    "mac": mac,
                    "blocked": True,
                    "reason": "DoS detection",
                    "unlock_time": state["unlock_time"]
                }
                self.policy_queue.put(policy)
                state["blocked"] = True
                state["counter"] = 0
```

Figure 3.1: Updated detection logic for time-based blocking.

It is crucial to note note how the blocking time in case of detection by the ML model is hardcoded. This is fundamental, since the model is also used to predict possible stealth attacks, in which the rate is very low, so using the threshold-exceeding logic could lead to incoherent ban times.
When blocking time expires, a policy with a remove request is placed, so the enforcement thread can delete the blocking rule (the same way it did in the previous version).

This change makes the mitigation mechanism more robust against persistent attacks, preventing attackers from being prematurely unblocked.

3.2 Policy Management Web-Applcation

Currently, network administrators can manage policies only through the CLI, for example by issuing *curl* commands to the controller's REST API. While functional, this approach is not user-friendly.

To simplify interaction, a web application has been developed on top of the REST API already provided by the controller. From an architectural point of view, the web application does not alter the internal logic of the controller. It simply acts as a REST client that sends HTTP requests (*GET*, *POST*, *DELETE*) to the API endpoints.

Below are some screenshots of the web-app's graphical interface.

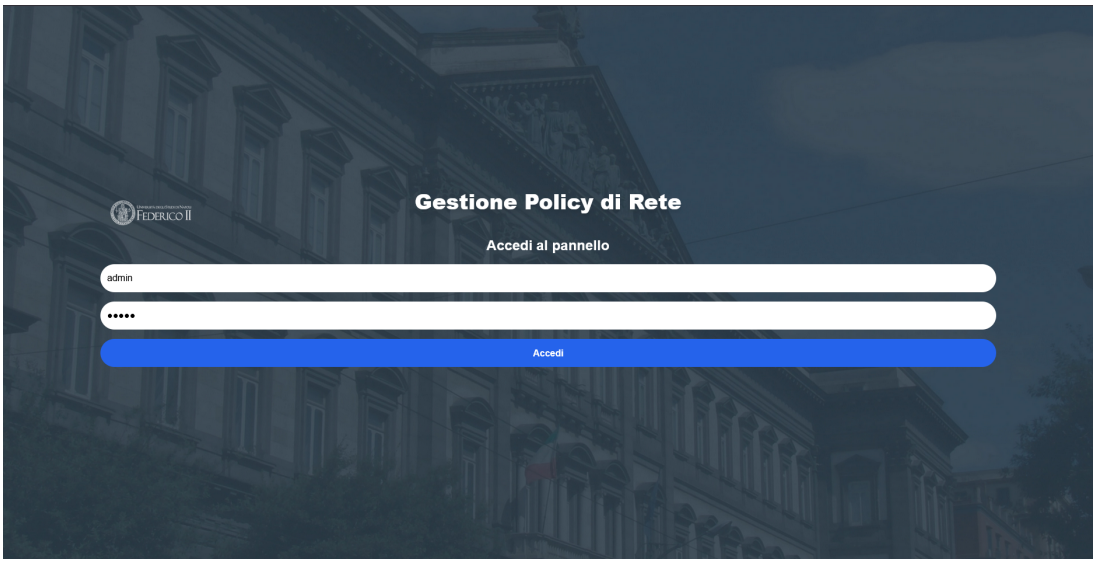


Figure 3.2: Login panel.

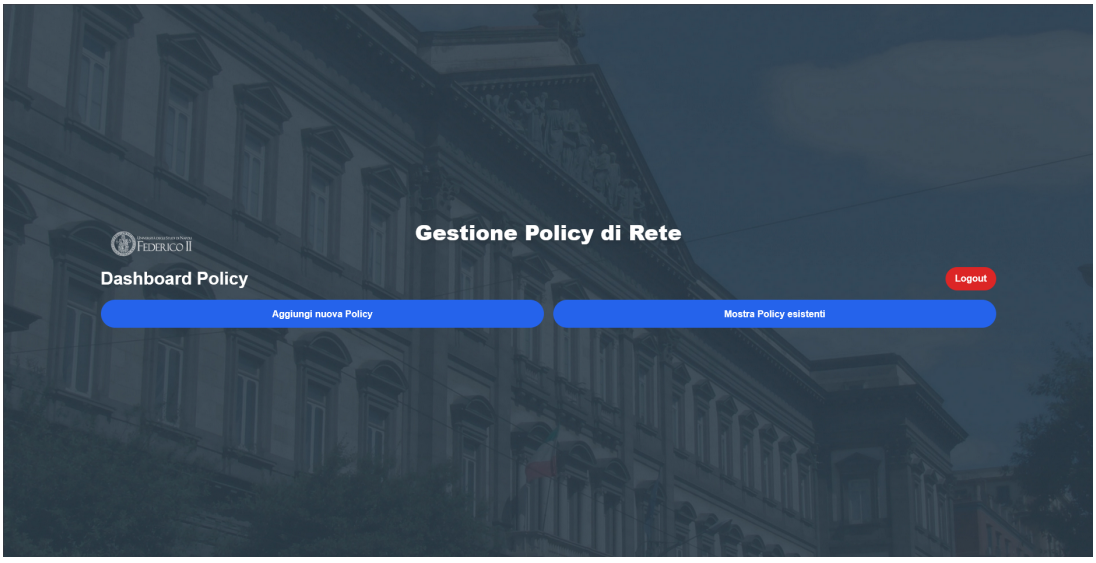


Figure 3.3: Main page.

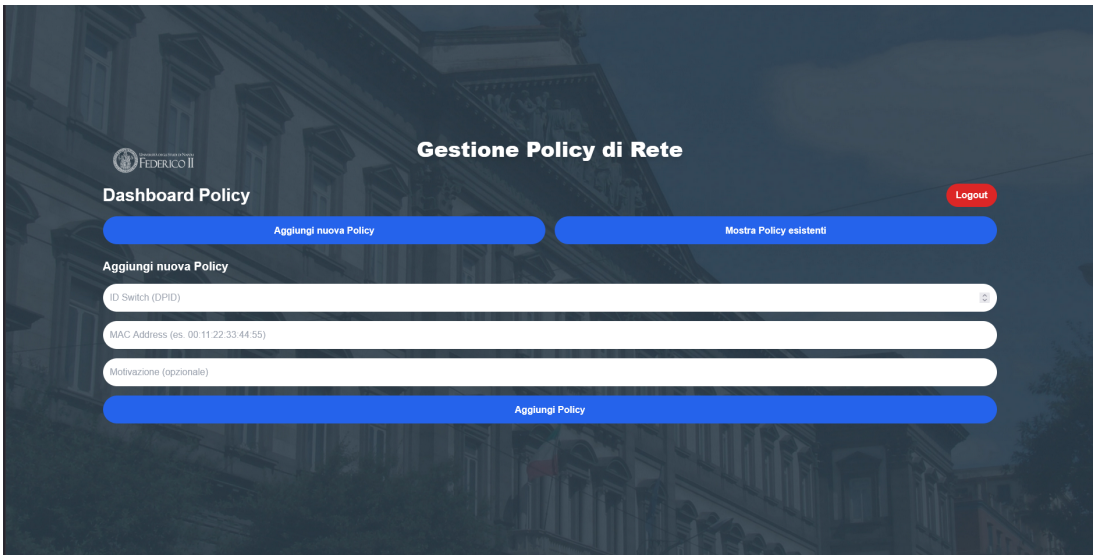


Figure 3.4: Add policy page.

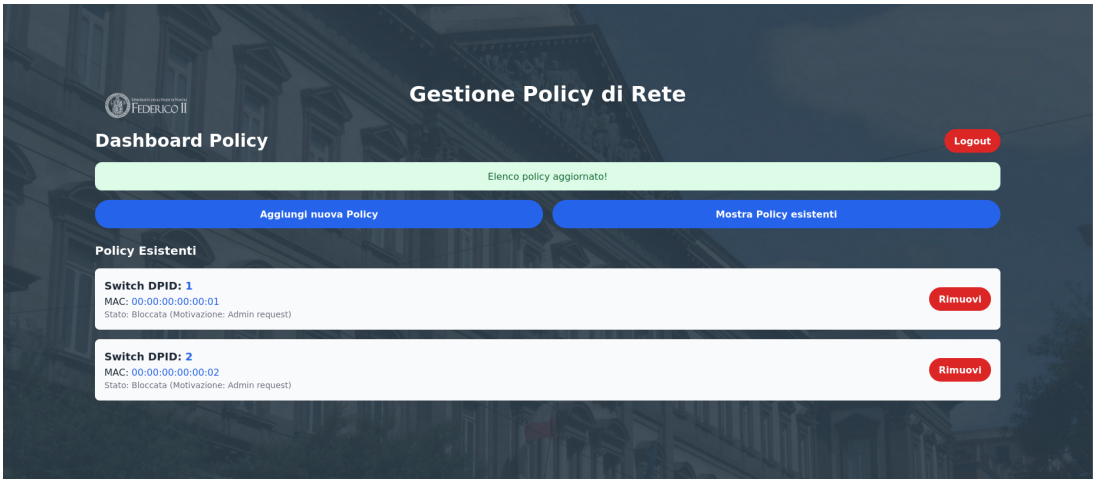


Figure 3.5: List and remove policy page.

Chapter 4

Evaluation

In this chapter, the proposed implementation of the controller was tested. In particular, section [4.2] and [4.3] respectively test classical and ML-based enforcement logic.

4.1 Normal Traffic Simulation

The following simulation is made to show the normal network functioning when normal traffic is generated. First of all, the *pingall* command was launched to check reachability between hosts. Then, both hosts *h1* and *h2* were configured to generate low-rate TCP traffic to host *h3*.

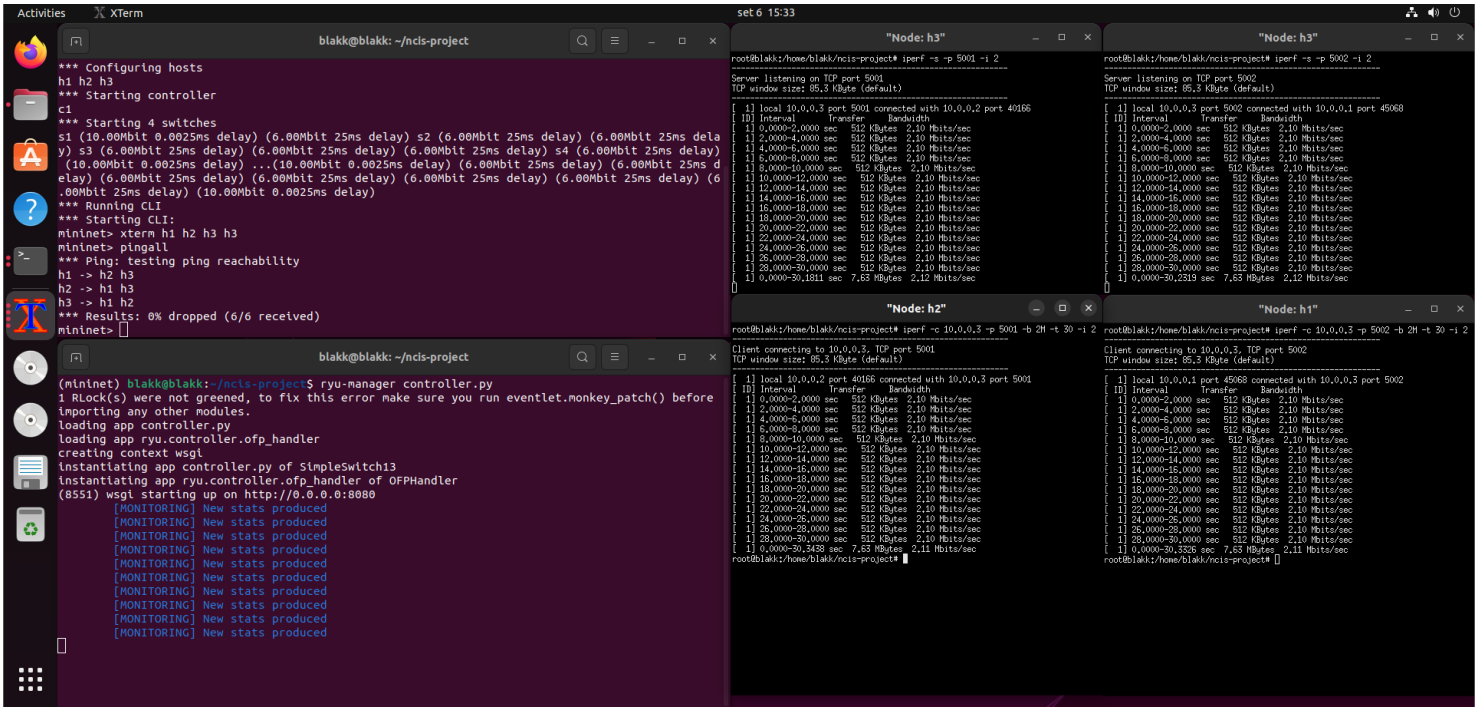


Figure 4.1: Normal traffic test.

4.2 Simple DoS Attack Simulation

In this section, the classic enforcement logic was tested (without using ML model), to test the controller's behavior against simple high-burst DoS attacks. To simulate the attack, host *h1* was configured to generate a large amount of UDP traffic towards host *h3*, causing congestion in the network. Meanwhile, host *h2* continued to generate normal TCP traffic towards host *h3*, allowing to observe the impact of the attack on TCP performance.

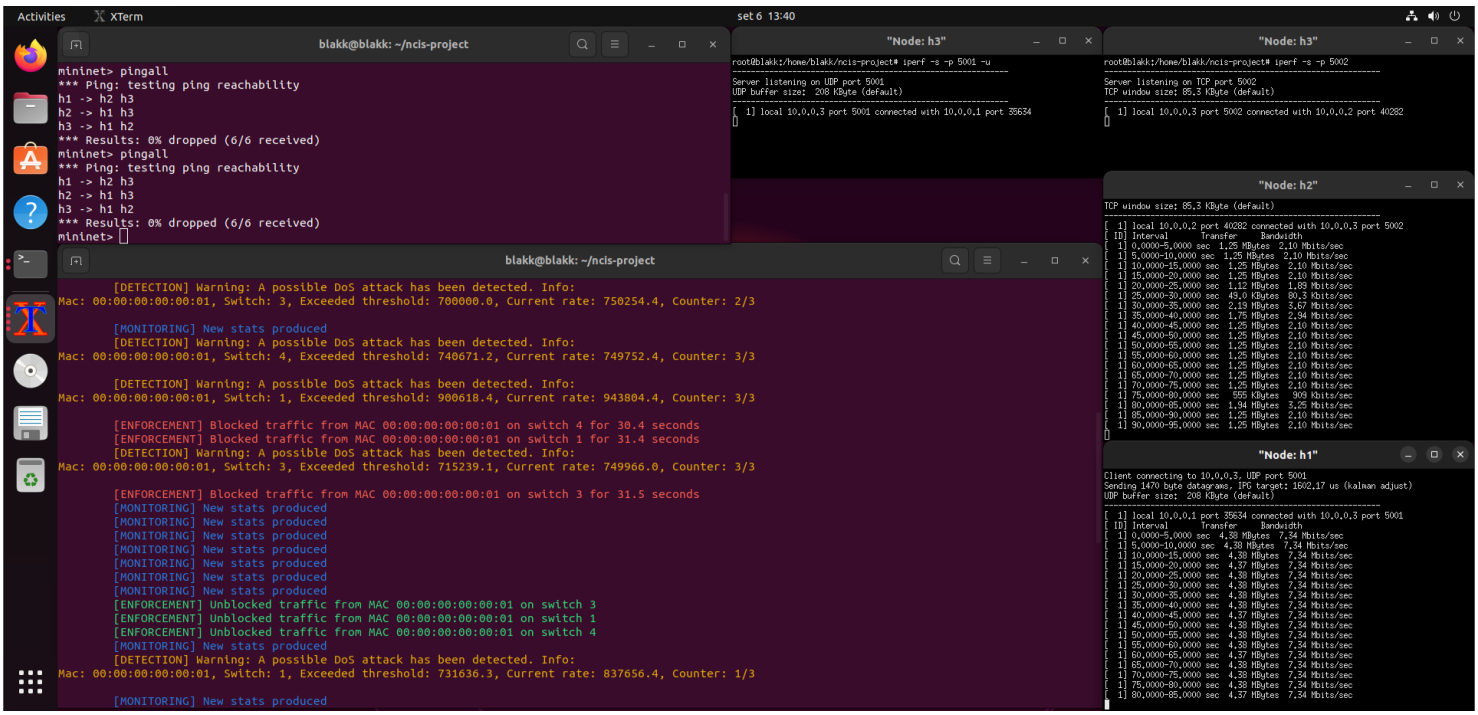


Figure 4.2: Simple DoS attack test.

As shown in the figure above, the attack is detected and the malicious host is blocked. Thanks to MAC address-level blocking policies, only host *h1* is blocked, while host *h2* continues to transmit freely.

4.3 Advanced DoS Attack Simulation

During this second simulation, the goal was to test the functioning of the ML model, to see how the controller reacts to distributed or more stealthy attacks. In particualr, two attacks were simulated.

Note: To better test the ML model's performance, the classic detection logic (based on rate and threshold) was temporarily disabled. Therefore, during the following simulations, only the ML model was running.

In the first attack, both hosts *h1* and *h2* were configured to generate large

amounts of UDP traffic towards host *h3*, simulating a Distributed Denial-of-Service (DDoS) attack.

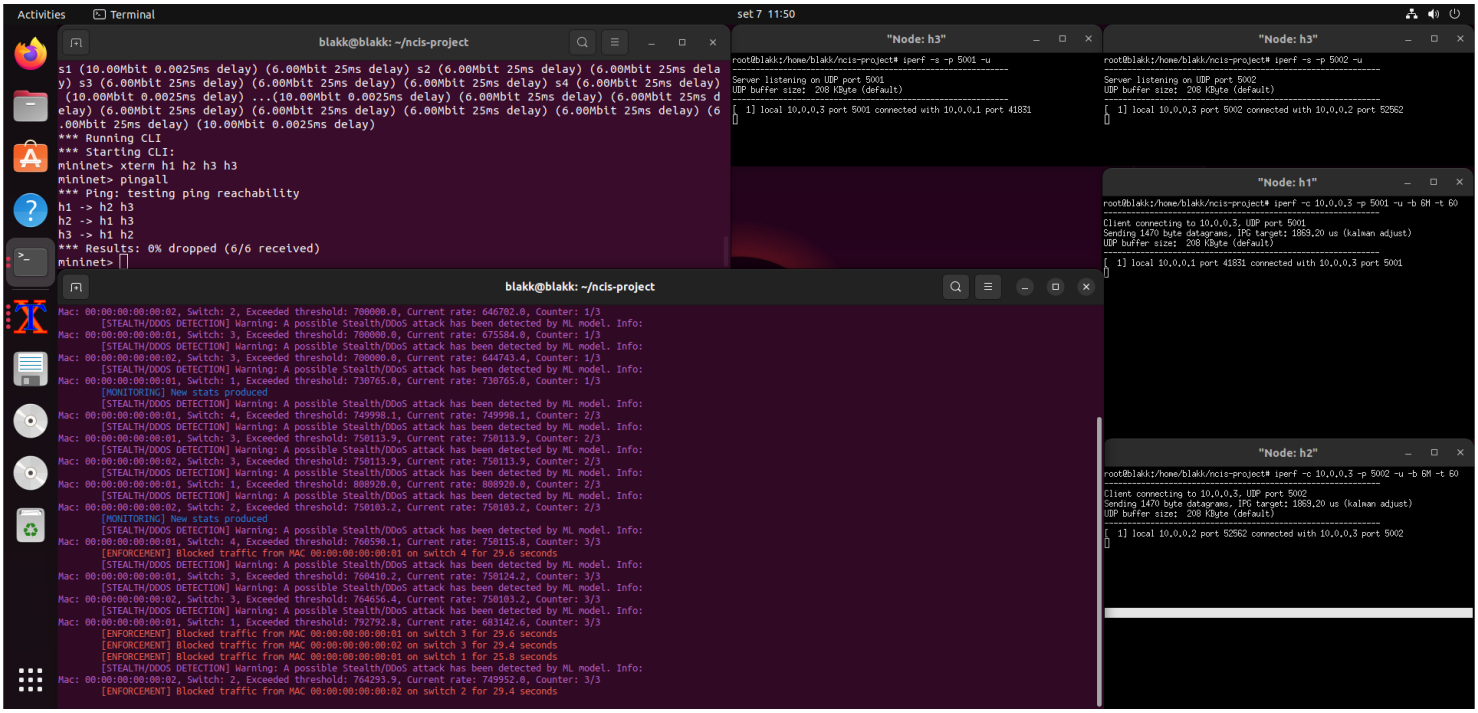


Figure 4.3: DDoS attack test.

Prima di mostrare il risultato della simulazione del secondo attacco, mostriamo l'impatto di un attacco stealth su un utente lecito della rete. Questo è utile per mostrare come agisce un attacco stealth, essendo questo diverso dal classico high-burst visto fino ad ora.

For the second simulation, host *h2* was configured to generate normal low-rate UDP traffic to *h3*, while host *h1* generated variable size and time ping packets, typical of stealth DoS attacks.

```bash
#!/bin/bash

TARGET=10.0.0.3

echo "[H1] Avvio attacco DoS Stealth"

while true; do
    SIZE=$((100 + RANDOM % 600))
    SLEEP=$(awk -v min=0.02 -v max=0.12 'BEGIN{s=min+rand()*(max-min); print s}')
    ping -c 1 -s $SIZE $TARGET > /dev/null
    sleep $SLEEP
done

echo "[H1] Attacco completato!"
```

Figure 4.4: Stealth script used by host *h1* to generate the stealth DoS.

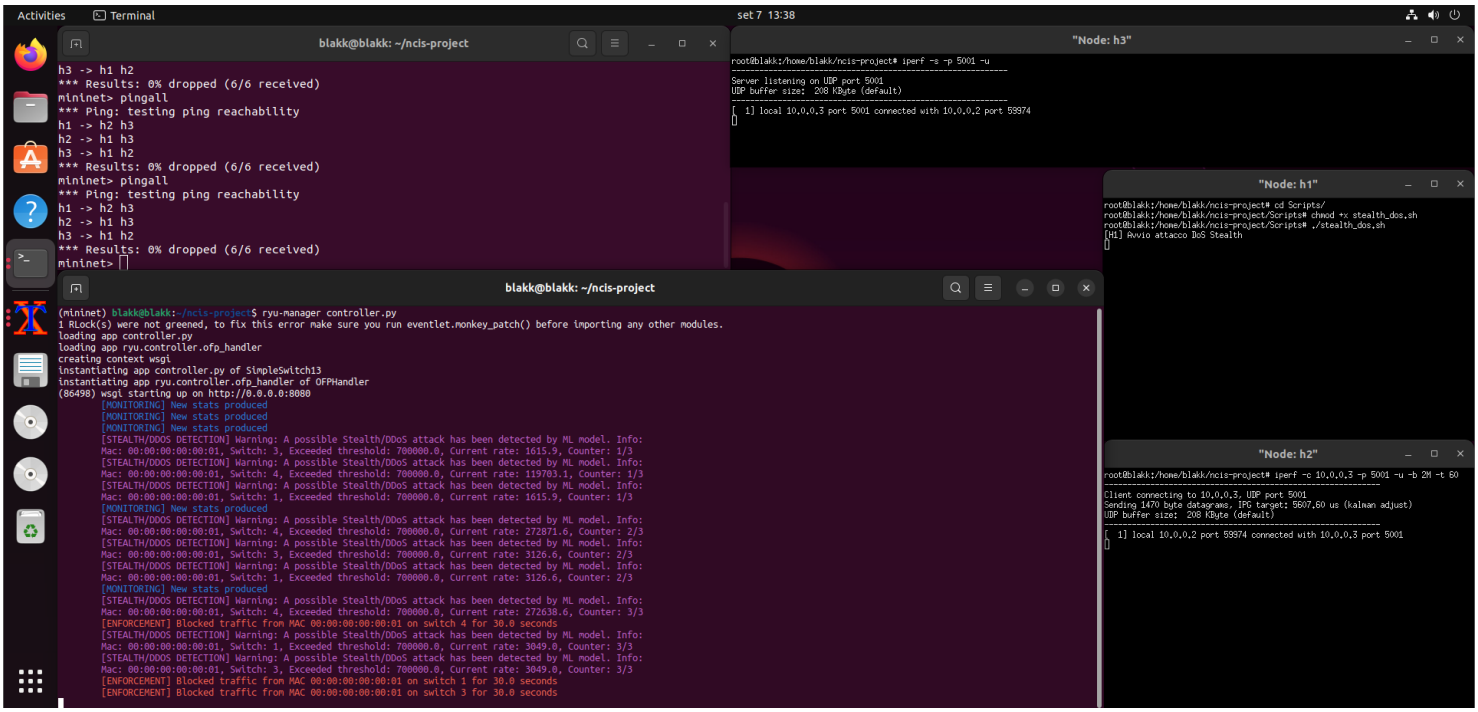


Figure 4.5: Stealth DoS attack test.

In both cases, the ML model was able to detect the attack attempt and block malicious hosts.

Chapter 5

Conclusions

This work extends the previous project by addressing several limitations in the detection and mitigation of DoS attacks in SDN. The introduction of dynamic thresholds, modular monitoring/detection/enforcement, per-host blocking, REST API and a Machine Learning model makes the system more flexible and accurate.

Despite these improvements, some limitations remain, discussed in section [5.2].

5.1 Controller Implementation

The final controller implementation is shown below. For the complete code of the entire project - including the various scripts used, the files related to the ML model and the web-app code - is available at the following link.

```
# === PARAMETRI GLOBALI ===
timeInterval = 10           # intervallo polling stats
X = 3                       # numero di cicli consecutivi sopra soglia per bloccare
STATIC_THRESHOLD = 7e5      # soglia statica, circa 80% della capacità critica dei link
K = 1                       # fattore di deviazione standard per la soglia dinamica
MEAN_RATIO = 1              # fattore di gestione media mobile
N_HISTORY = 20              # numero di campioni per valutare soglia dinamica e raccolta dati
ALPHA = 0.8                 # decremento contatore se rate < 80% della soglia attuale
BAN_RATIO = 10              # fattore di ban-time
MIN_RATE = 250              # rate minimo sotto cui non far lavorare il modello di ML
BAN_TIME = 30               # tempo di ban solo per attacchi stealth e ddos

# MAC noti degli host
HOST_MACS = {"00:00:00:00:00:01", "00:00:00:00:00:02", "00:00:00:00:00:03"}
FEATURE_CLASS = "attacco"

# colori per output console
RED = "\033[91m"
GREEN = "\033[92m"
YELLOW = '\033[93m'
BLUE = '\033[94m'
PURPLE = '\033[95m'
CYAN = '\033[96m'
RESET = "\033[0m"

class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    _CONTEXTS = {
        'wsgi': WSGIApplication
    }

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.mac_to_port = {}

        # strutture dati per statistiche
        self.port_stats = {}          # {dpid: {port: {"rx_bytes": ..., "rx_packets": ...}}}
        self.prev_port_stats = {}     # stats al ciclo precedente

        self.detection_state = {}     # {dpid: {mac: {"blocked": False, "counter": 0}}}
        self.rate_history = {}        # {dpid: {mac: [rate1, rate2,...]}}
        self.policies = {}            # {dpid: {mac: policy}}

        # setup per raccolta istanze (ML)
        self.mac_timestamps = {}      # {dpid: {mac: [t1, t2, ...]}}
        self.dst_sources = {}         # {dpid: {dst: set([src1, src2, ...])}}
        self.mac_pkt_sizes = {}       # {dpid: {mac: [pkt_len1, pkt_len2, ...]}}
        self.csv_file = "traffic.csv"
        self.csv_fields = [
            "timestamp","dpid","mac",
            "mean_rate","var_rate","max_rate","min_rate",
            "mean_interval","std_interval","burst_count",
            "src_diversity","small_pkt_ratio","class"
        ]
        # creazione del file csv
        '''
        if not os.path.exists(self.csv_file) or os.path.getsize(self.csv_file) == 0:
            with open(self.csv_file, "w", newline="") as f:
                writer = csv.DictWriter(f, fieldnames=self.csv_fields)
                writer.writeheader()
        '''
        # setup modello (RandomForest)
        self.ml_model = joblib.load("dos_ddos_detector.pkl")
        self.ml_model.n_jobs = 1
        self.ml_features = [
            "mean_rate","var_rate","max_rate","min_rate",
            "mean_interval","std_interval","burst_count",
            "src_diversity","small_pkt_ratio"
        ]

        # setup REST per web-app
        wsgi = kwargs['wsgi']
        wsgi.register(PolicyAPI, {'controller': self})

        # code thread-safe per comunicazione tra thread
        self.stats_queue = queue.Queue()     # monitoring → detection
        self.policy_queue = queue.Queue()    # detection → enforcement

        # spawn dei thread
        self.monitor_thread = hub.spawn(self._monitor_loop)
        self.detect_thread = hub.spawn(self._detect_loop)
        self.enforce_thread = hub.spawn(self._enforce_loop)
        # self.feature_thread = hub.spawn(self._feature_loop)   # thread di raccolta dati per addestrare il modello
```

Figure 5.1: Global parameters and initialization.

```python
# === MONITORING ===
@set_ev_cls(ofp_event.EventOFPStateChange, [MAIN_DISPATCHER, DEAD_DISPATCHER])
def _state_change_handler(self, ev):
    dp = ev.datapath
    if ev.state == MAIN_DISPATCHER:
        self.datapaths[dp.id] = dp
    elif ev.state == DEAD_DISPATCHER:
        self.datapaths.pop(dp.id, None)

def _request_stats(self, datapath):
    self.logger.debug('send stats request: %016x', datapath.id)
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
    req = parser.OFPPortStatsRequest(datapath, 0, ofproto.OFPP_ANY)
    datapath.send_msg(req)

def _monitor_loop(self):
    while True:
        for dp in self.datapaths.values():
            self._request_stats(dp)
        hub.sleep(timeInterval)
        self.logger.info(BLUE + '\t[MONITORING] New stats produced' + RESET)

@set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
def _port_stats_reply_handler(self, ev):
    dp = ev.msg.datapath
    dpid = dp.id
    self.port_stats.setdefault(dpid, {})
    self.prev_port_stats.setdefault(dpid, {})

    # aggiorna le statistiche correnti con i dati del reply
    for stat in ev.msg.body:
        port_no = stat.port_no
        self.port_stats[dpid][port_no] = stat

    # calcola il rate di traffico per ogni porta
    for port_no, stat in self.port_stats[dpid].items():
        if port_no in self.prev_port_stats[dpid]:
            prev_stat = self.prev_port_stats[dpid][port_no]
            # calcola la differenza in byte e il tempo trascorso
            delta_bytes = stat.rx_bytes - prev_stat.rx_bytes
            delta_time = timeInterval

            # calcola il rate in byte/s
            rate = (delta_bytes / delta_time) if delta_time > 0 else 0

            # trova il MAC address associato a questa porta
            # si assume una topologia semplice con 1 host per porta,
            # e si usa la mappa mac_to_port per la ricerca inversa.
            mac_to_check = None
            for mac, port in self.mac_to_port.get(dpid, {}).items():
                if port == port_no and mac in HOST_MACS:
                    mac_to_check = mac
                    break

            if mac_to_check:
                # usa la storia dei rate per calcolare la soglia dinamica
                self.rate_history.setdefault(dpid, {}).setdefault(mac_to_check, []).append(rate)
                hist = self.rate_history[dpid][mac_to_check]
                if len(hist) > N_HISTORY:
                    hist.pop(0)

                if hist:
                    mean_val = np.mean(hist)
                    std_val = np.std(hist)
                    dynamic_threshold = mean_val * MEAN_RATIO + K * std_val
                else:
                    dynamic_threshold = STATIC_THRESHOLD

                # push delle stats in coda detection
                self.stats_queue.put((dpid, mac_to_check, rate, dynamic_threshold))

    # aggiorna le statistiche precedenti con le statistiche correnti per il prossimo ciclo
    self.prev_port_stats[dpid] = self.port_stats[dpid].copy()
```

Figure 5.2: Monitoring.

```
# === DETECTION ===
def _detect_loop(self):
    while True:
        try:
            dpid, mac, rate, dynamic_threshold = self.stats_queue.get(timeout=timeInterval)
        except queue.Empty:
            continue

        now = time.time()

        # gestisci eventuali sblocchi
        for dp_id, hosts in self.detection_state.items():
            for mac_addr, state in hosts.items():
                if state.get("blocked") and now >= state.get("unlock_time", 0):
                    self.policy_queue.put({
                        "action": "remove",
                        "switch": dp_id,
                        "mac": mac_addr
                    })
                    state["blocked"] = False
                    state["counter"] = 0  # reset contatore dopo sblocco

        # init stato se non presente
        self.detection_state.setdefault(dpid, {}).setdefault(mac, {
            "blocked": False,
            "counter": 0,
            "unlock_time": 0
        })
        state = self.detection_state[dpid][mac]

        threshold = max(STATIC_THRESHOLD, dynamic_threshold)

        if not state["blocked"]:
            # --- detection classica ---
            classical_detect = False
            classical_detect = rate > threshold
            # --- detection machine learning ---
            ml_detect = False
            if mac in HOST_MACS:    # evita predizioni su dati degli switch (solo host → host)
                features = self.extract_features(dpid, mac)
                if features:
                    # evita predizione su dati incompleti
                    if rate > MIN_RATE and any(features[f] > 0 for f in self.ml_features):
                        X_input = pd.DataFrame([[features[f] for f in self.ml_features]],
                                               columns=self.ml_features)
                        try:
                            pred = self.ml_model.predict(X_input)
                            # supponendo: 0 = attacco, 1 = normale
                            if pred[0] == 0:
                                ml_detect = True
                        except Exception as e:
                            hub.sleep(1)

            if classical_detect:
                state["counter"] = min(X, state["counter"] + 1)
                self.logger.info(YELLOW + f"\n\t[DETECTION] Warning: A possible DoS attack has been detected. Info:"
+ RESET)
                self.logger.info(YELLOW + f"Mac: {mac}, Switch: {dpid}, Exceeded threshold: {threshold:.1f}, Current
rate: {rate:.1f}, Counter: {state['counter']}/{X}" + RESET)
            if ml_detect:
                state["counter"] = min(X, state["counter"] + 1)
                self.logger.info(PURPLE + f"\n\t[STEALTH/DDOS DETECTION] Warning: A possible Stealth/DDoS attack has
been detected by ML model. Info:" + RESET)
                self.logger.info(PURPLE + f"Mac: {mac}, Switch: {dpid}, Exceeded threshold: {threshold:.1f}, Current
rate: {rate:.1f}, Counter: {state['counter']}/{X}" + RESET)

            # decrementa contatore se non riconosciuto da modello ML
            # (altrimenti gli attacchi stealth non vengono bloccati)
            if rate < threshold * ALPHA and not ml_detect:
                state["counter"] = max(0, state["counter"] - 1)

            # blocco se il contatore raggiunge X
            if state["counter"] >= X:
                if ml_detect:
                    block_time = BAN_TIME       # tempo di blocco fisso per evitare problemi con attacchi stealth
                else:
                    excess_ratio = rate / threshold
                    block_time = excess_ratio * BAN_RATIO * X
                state["unlock_time"] = now + block_time
                policy = {
                    "switch": dpid,
                    "mac": mac,
                    "blocked": True,
                    "reason": "DoS detection",
                    "unlock_time": state["unlock_time"]
                }
                self.policy_queue.put(policy)
                state["blocked"] = True
                state["counter"] = 0
```

Figure 5.3: Detection.

```
# === ENFORCEMENT ===
def _enforce_loop(self):
    while True:
        try:
            policy = self.policy_queue.get(timeout=1)
        except queue.Empty:
            continue

        dpid = str(policy.get("switch"))
        mac = policy.get("mac")
        reason = str(policy.get("reason", "manual"))

        if policy.get("action") == "remove":
            if dpid in self.policies and mac in self.policies[dpid]:
                del self.policies[dpid][mac]
            dp = self.datapaths.get(int(dpid))
            if dp:
                self.unlock_flow(dp, mac)
            continue

        if "blocked" not in policy:
            policy["blocked"] = True
            policy["reason"] = reason

        self.policies.setdefault(dpid, {})[mac] = policy
        dp = self.datapaths.get(int(dpid))
        if dp and policy["blocked"]:
            block_duration = policy["unlock_time"] - time.time()
            self.lock_flow(dp, mac, block_duration)

def lock_flow(self, dp, mac, block_time):
    parser = dp.ofproto_parser
    ofproto = dp.ofproto
    match = parser.OFPMatch(eth_src=mac)
    mod = parser.OFPFlowMod(datapath=dp, priority=2, match=match, instructions=[],
                            command=ofproto.OFPFC_ADD, out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
    dp.send_msg(mod)
    print(RED + f"\t[ENFORCEMENT] Blocked traffic from MAC {mac} on switch {dp.id} for {block_time:.1f} seconds" +
RESET)

def unlock_flow(self, dp, mac):
    parser = dp.ofproto_parser
    ofproto = dp.ofproto
    match = parser.OFPMatch(eth_src=mac)
    mod = parser.OFPFlowMod(datapath=dp, priority=2, match=match,
                            command=ofproto.OFPFC_DELETE, out_port=ofproto.OFPP_ANY, out_group=ofproto.OFPG_ANY)
    dp.send_msg(mod)
    print(GREEN + f"\t[ENFORCEMENT] Unblocked traffic from MAC {mac} on switch {dp.id}" + RESET)
```

Figure 5.4: Enforcement.

```
# === MACHINE LEARNING MODEL ===
def extract_features(self, dpid, mac):
    rates = self.rate_history.get(dpid, {}).get(mac, [])
    times = self.mac_timestamps.get(dpid, {}).get(mac, [])

    if len(rates) < 1 or len(times) < 1:
        return None  # non abbastanza dati

    mean_rate = np.mean(rates)
    var_rate = np.var(rates)
    max_rate = np.max(rates)
    min_rate = np.min(rates)

    intervals = np.diff(times)
    mean_interval = np.mean(intervals) if len(intervals) > 0 else 0
    std_interval = np.std(intervals) if len(intervals) > 0 else 0

    # burst count: quante volte il rate supera 1.5x la media
    burst_count = sum(r > mean_rate * 1.5 for r in rates)

    # numero di sorgenti per la stessa destinazione
    # (prende la media dei sorgenti unici visti nelle destinazioni contattate da questo MAC)
    dst_map = self.dst_sources.get(dpid, {})
    src_diversity = 0
    if dst_map:
        counts = [len(srcs) for srcs in dst_map.values()]
        if counts:
            src_diversity = sum(counts) / len(counts)

    # percentuale pacchetti piccoli (<100B)
    pkt_sizes = self.mac_pkt_sizes.get(dpid, {}).get(mac, [])
    small_pkt_ratio = 0
    if pkt_sizes:
        small_pkt_ratio = sum(1 for s in pkt_sizes if s < 100) / len(pkt_sizes)

    features = {
        "timestamp": time.time(),
        "dpid": dpid,
        "mac": mac,
        "mean_rate": mean_rate,
        "var_rate": var_rate,
        "max_rate": max_rate,
        "min_rate": min_rate,
        "mean_interval": mean_interval,
        "std_interval": std_interval,
        "burst_count": burst_count,
        "src_diversity": src_diversity,
        "small_pkt_ratio": small_pkt_ratio,
        "class": str(FEATURE_CLASS)
    }
    return features

def _feature_loop(self):
    while True:
        hub.sleep(timeInterval)  # ogni timeInterval estrai e salva
        for dpid, macs in self.rate_history.items():
            for mac in macs.keys():
                if mac not in HOST_MACS:  # salta MAC non host
                    continue
                features = self.extract_features(dpid, mac)
                if features:
                    features["class"] = str(FEATURE_CLASS)  # aggiungi colonna classe
                    with open(self.csv_file, "a", newline="") as f:
                        writer = csv.DictWriter(
                            f,
                            fieldnames=self.csv_fields,
                            extrasaction="ignore",
                            restval=""
                        )
                        writer.writerow(features)
        print(CYAN + f"\t[FEATURE EXTRACTOR] New features extracted" + RESET)
```

Figure 5.5: ML model.

5.2 Future improvements

In addition to some of the possible vulnerabilities reported in the sections, the proposed controller version presents a series of possible improvements postponed to future implementations. Among the many, those that emerged from testing and I believe are the most important are:

- *Exponential backoff*: The block time depends exclusively on how much the rate exceeds the threshold, so an attacker who persistently tries (after the unban) to attack the network is not punished enough. If the same attacker retries the attack after being unbanned, then the ban time should increases exponentially with the number of attempts.

- *Persistent blocking logic*: Currently, bans are only handled at runtime, meaning that if the controller crashes it "forgets" who was banned. A file storing the applied policies could be added, which should then be loaded when the controller boots.

- *TCP flag control*: Currently, detection logic is limited to MAC addresses only. It may be useful to also check for TCP flags used in DoS attacks, such as SYN Floods and ACK Floods.

- *Duplicate rules*: Whenever a host is detected as malicious, a blocking rule is enforced on every switch where the host's rate exceeded the threshold. The blocking rule could be enforced only on the first switch along the path, to avoid duplicate rules that waste resources.

- *Link-dependent static threshold*: The static threshold used is unique and is chosen based on the "bottleneck" connection of the network. Rather than using just one, a list of static thresholds that depend on the link between switches could be used.