Customizable Firewall Control in Software-Defined Networks

A Campus LAN Simulation with Proxy ARP

Gina K. Nasseri

Abstract

This paper demonstrates using software defined networks (SDNs) to implement a firewall that can be centrally controlled and programmed to customize different levels of securities for different local area networks (LANs) and demonstrates the use of a SDN Proxy ARP to optimize the firewall set-up time.

1 Introduction

As technology advances and the diversity of devices continues to explode, software defined networks (SDNs) are becoming increasingly more necessary. The rapid evolution of applications can be largely attributed to the advent of the operating system, which provides a consistent layer of abstraction between applications and the underlying hardware, enabling development independent of the device specifications. SDNs apply this same concept to networking by providing a layer of abstraction that allows applications, services, and configurations to be designed independently of the underlying hardware. Within this framework, firewalls emerge as a natural extension. The programmability of SDNs enables firewall policies to be customized and centrally managed through the controller, rather than requiring individual configuration on each device as in a traditional network. This paper presents a simulated example of how a university campus could implement a centralized firewall application to control access levels for its LANs. It also discusses the implementation and demonstrates how to optimize firewall setup time by using a technique called Proxy ARP.

2 Background

2.1 Motivation

In office and campus environments, networks typically support a wide range of devices. Even a single user may operate multiple endpoints, such as a laptop, phone, or desktop computer. In such settings, access restrictions are often required for both security and policy compliance. The complexity increases further when multiple access levels must be enforced, as in government offices where distinct networks may be subject to different security clearances and restrictions. In traditional architectures, implementing these rules requires configuring each switch individually, a process which is meticulous and time consuming. In contrast, an SDN firewall operates through the centralized controller, which maintains a global view of devices and links. Firewall policies can therefore be installed, updated, and managed consistently across the entire network from a single point.

For this project, the goal was to demonstrate how an SDN firewall can be used to enforce different security levels across multiple networks. The motivation was drawn from real-world environments, such as government agencies, where a single overall network may be divided into multiple LANs or VLANs, each designated with a specific clearance level. To model this in a simpler way, the simulated environment was based on a university campus with two well-defined clearance levels. An ADMIN server is given privileged access to all devices on the network and has the ability to enable or disable the firewall, while STUDENT LANs are restricted to communicating only with devices within their own LAN.

2.2 Network Topology

The network topology is shown in figure 1 (adapted from Montazerolghaem [1].), it uses a three-tier topology consisting of a core layer, an aggregation layer, and an access layer. This hierarchical design is common in both campus and data center networks. The core switch provides high-speed connectivity between aggregation switches, which in turn connect to the access layer switches. Each access switch serves as the entry point for a local area network (LAN). In

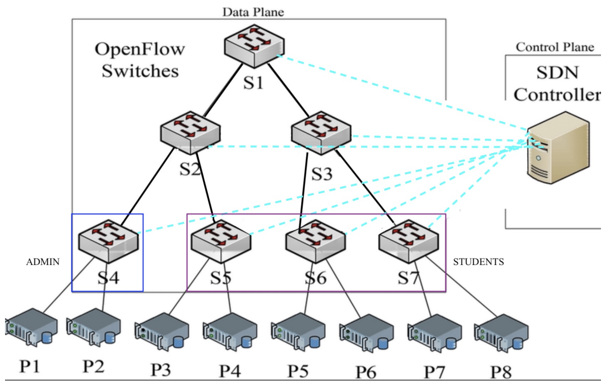


Figure 1: The simulated network.

this project, four access switches are used: one designated as the Admin LAN and three as Student LANs. This structure allows clear separation of roles for evaluating firewall policies across different LANs, while remaining simple enough to simulate in Mininet, a lightweight network emulator that enables the creation of custom topologies and the deployment of SDN controllers using the OpenFlow protocol within a virtual environment.

2.3 SDN Controller and OpenFlow

At the core of a software defined network is the controller, which serves as the centralized decision-making component. Unlike traditional networks where each switch makes forwarding decisions independently, an SDN controller maintains a global view of the network and installs forwarding rules on the switches under its control. This separation of the control plane from the data plane, also shown in figure 1, is what allows SDNs to be highly flexible and programmable.

OpenFlow is the standard protocol that enables communication between the controller and the switches. Through OpenFlow, the controller can install, update, and remove flow entries on each switch. These flows specify how packets should be handled, based on fields such as source and destination MAC addresses, IP addresses, or transport layer ports. By programming these flows centrally, the controller can implement network-wide policies such as routing, load balancing, or firewalling without the need to configure each device individually.

3 Method

3.1 Mininet Implementation

This project was implemented and tested using Mininet, a lightweight network emulator that provides a virtual environment for creating custom topologies and running SDN controllers. Mininet runs in Python, and the controller used here is POX, an open-source framework that communicates with switches using the OpenFlow protocol. Two custom modules were developed for this project: `proxy_learning.py`, which implements a learning algorithm with proxy ARP, and `simple_firewall.py`, which implements the firewall and its control logic. The source code for both modules is included in the Appendix and was also submitted along with the report.

Learning Algorithm

For the ADMIN to manage the network, it must first establish the location of all devices before firewall policies can be applied. This initial process functions as a bootstrap phase, run once at system startup, during which all devices are temporarily able to communicate freely. This ensures that the ADMIN can reach every host and, just as importantly, that the controller can learn all device locations in order to install the correct firewall rules. Once this phase is complete, the firewall is enabled. The Python code for the implementation is provided in the Appendix (and was submitted along with the report in the file `proxy_learning.py`).

To support this, the controller implements a custom learning algorithm that tracks host locations and suppresses unnecessary broadcasts. Each switch is associated with a `SwitchState` object that maintains the mapping between MAC addresses and switch ports. When a host joins the network or changes ports, the controller updates the mapping accordingly. These mappings are derived both from host events provided by POX's host tracker and from packets received at the switch. To reduce broadcast traffic, the controller also implements a proxy ARP mechanism, where, whenever an ARP request is received, the controller checks its global `ip_to_mac` table. If the mapping is already known, the controller constructs a synthetic ARP reply using the stored MAC and IP, encapsulates it in an Ethernet frame, and sends it directly back to the requester. This avoids waiting for the destination host to respond and removes the need to flood the request across the network, significantly reducing startup time and network traffic.

Firewall

The firewall is implemented with hardcoded knowledge of the network topology, including the set of switches and their attached devices. No restrictions are applied to the ADMIN switch, while each Student switch receives rules that enforce isolation. A high-priority rule allows communication only between hosts on the same Student LAN, followed by lower-priority rules that block external traffic. Additional rules permit incoming traffic from the ADMIN LAN to the Student LANs, reflecting the privileged role of the administrator. The Python code for the implementation is provided in the Appendix (and was submitted along with the report in the file `admin_firewall.py`).

The firewall can be toggled on and off by the ADMIN. At startup, the system enters a bootstrap phase to allow all hosts to learn each other's locations. The firewall is then activated automatically, and the ADMIN can later disable it by sending a TCP signal to a designated port. While this signalling method would not be used in a production network, it provides a straightforward mechanism for demonstrating administrative control of the SDN controller in this simulation.

To disable the firewall, a trigger flow listens for traffic on TCP port 8888. When this signal is received, the controller deletes firewall rules from the switch tables. In newer versions of OpenFlow, flow entries can be grouped and removed selectively based on their cookie values. In this project, cookie values were still assigned to the firewall rules so they could be observed in the tables, but OpenFlow 1.0 in POX provides fewer match options, making selective deletion unreliable. The simplest approach was to delete all flows and then call the same routine used at `ConnectionUp`, which re-installs the table-miss rule and the trigger rules for TCP ports 9999 and 8888. Proxy ARP reduces the cost of relearning after deletion, so this method worked well for the simulation even if a finer-grained approach would be used with a newer controller.

Bootstrap

The network is initialized with a bootstrap phase to ensure host discovery before the firewall is enabled. This is done by running a single `pingAll`, which allows the controller to learn the location of all hosts and populate its tables. The bootstrap is followed by the activation of the firewall triggers, where designated ADMIN hosts listen on specific TCP ports (9999 for ON, 8888 for OFF). This phase guarantees that the controller has full knowledge of the network before firewall policies are enforced.

4 Testing Procedures

4.1 Firewall Tests

The firewall configuation and toggle feature were tested using three methods:

1. **Interactive testing with Mininet CLI.** The controller was launched with the `proxy_learning` and `admin_firewall` modules, and the network started with the bootstrap script. The script first runs `pingAll` to establish connectivity, then activates the firewall. From the Mininet CLI, the firewall was tested by running `pingAll`, then toggling it by sending TCP signals from the ADMIN hosts (`8888` to disable, `9999` to re-enable) and repeating `pingall` with the firewall on and off. Additionally, the controller output was inspected to verify the log messages indicating when firewall rules were installed and deleted.

2. **Flow table inspection.** OpenFlow rules were examined with `ovs-ofctl`. Flow tables were recorded after bootstrap, after firewall activation, after deletion of flows, after a subsequent `pingAll`, and after re-enabling the firewall. flow tables were inspected using ovs-ofctl, the Open vSwitch command-line tool for querying and managing OpenFlow rules on switches

3. **Wireshark observation.** Traffic was captured on Student-facing switch ports, such as `s6-eth3` connected to H6. Pings from an ADMIN host and from another Student host were compared with the firewall disabled, enabled, and disabled again to observe enforcement of isolation rules.

4.2 Proxy ARP Tests

Proxy ARP performance was tested using two methods:

1. **Controller output inspection.** With all learning tables initially empty, select two hosts, h_i and h_j. Have h_i ping h_j so the controller can learn the mapping for h_i. Then select a third host, h_k, to ping h_i. If Proxy ARP is enabled and functioning correctly, the controller should generate a direct ARP reply to h_k, whereas without Proxy ARP, h_k's ARP request will be broadcast across the network. Additionally, running `pingAll` with Proxy ARP enabled should produce noticeably fewer ARP broadcasts compared to when it is disabled.

2. **Wireshark capture.** To quantify the effect, Wireshark was used to capture traffic during `pingAll` with Proxy ARP enabled and disabled. Separate captures were taken on the up link of the aggregate switch S2 as well as a capture listening to all S2 ports. Metrics such as packet counts, broadcast rates, and completion times were collected for later comparison.

5 Results

5.1 Firewall Test Results

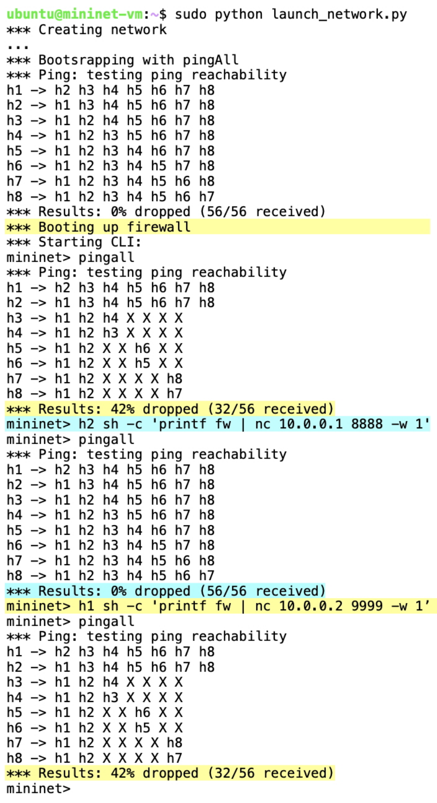
1. Mininet CLI and Controller Output

The Mininet and POX controller outputs in Figure 2 show a full cycle of enabling and disabling the firewall. The `launch_network.py` script first opens the designated TCP listening ports on the ADMIN hosts, runs a bootstrap phase with `pingAll` ("***Bootstrapping with pingAll"), then signals the firewall to turn on ("***Booting up firewall," highlighted in yellow) before handing control to the Mininet CLI. In the controller log (Figure 2b), the trigger is detected (first yellow line) and the isolation rules are installed. Back in the CLI, the next `pingAll` confirms the rules are active: Student LANs are isolated from each other, but still reach the ADMIN and their local peers. The X's in the output, along with the 42% packet drop summary, make the effect clear.

The disabling mechanism was then tested. The ADMIN sends a TCP signal to port 8888 on the other ADMIN host, which the controller picks up (blue line in Figure 2b), deleting all firewall rules. A subsequent `pingAll` shows connectivity restored, with the controller logs indicating the learning algorithm repopulating the tables. To re-enable, a TCP signal is sent to port 9999, the same command used at startup. The controller log again shows the trigger detected and the isolation rules reinstalled (yellow), and `pingAll` confirms the Student LANs are once again isolated. In short, the bootstrap, enable, and disable mechanisms all worked as designed.

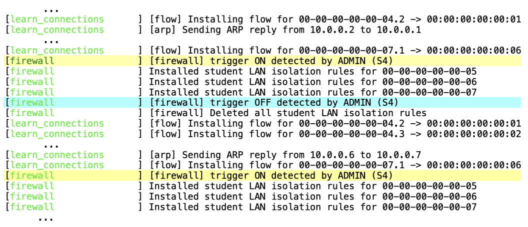
2. Flow Table Inspection

Flow table inspection further verified the controller's behaviour. The following figures show the OpenFlow tables for an ADMIN switch (S4) and a Student switch (S5) at different stages of the cycle. For readability, unused columns have been removed and only table updates are shown.



```
ubuntu@mininet-vm:~$ sudo python launch_network.py
*** Creating network
...
*** Bootsrapping with pingAll
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
*** Booting up firewall
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 X X X X
h4 -> h1 h2 h3 X X X X
h5 -> h1 h2 X X h6 X X
h6 -> h1 h2 X X h5 X X
h7 -> h1 h2 X X X X h8
h8 -> h1 h2 X X X X h7
*** Results: 42% dropped (32/56 received)
mininet> h2 sh -c 'printf fw | nc 10.0.0.1 8888 -w 1'
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
mininet> h1 sh -c 'printf fw | nc 10.0.0.2 9999 -w 1'
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 X X X X
h4 -> h1 h2 h3 X X X X
h5 -> h1 h2 X X h6 X X
h6 -> h1 h2 X X h5 X X
h7 -> h1 h2 X X X X h8
h8 -> h1 h2 X X X X h7
*** Results: 42% dropped (32/56 received)
mininet>
```

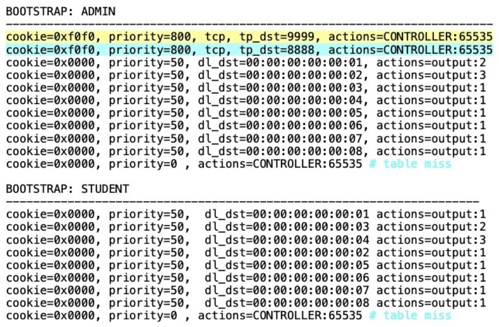
(a) Mininet CLI output.



```
      ...
[learn_connections      ] [flow] Installing flow for 00-00-00-00-00-04.2 -> 00:00:00:00:00:01
[learn_connections      ] [arp] Sending ARP reply from 10.0.0.2 to 10.0.0.1
      ...
[learn_connections      ] [flow] Installing flow for 00-00-00-00-00-07.1 -> 00:00:00:00:00:06
[firewall               ] [firewall] trigger ON detected by ADMIN (S4)
[firewall               ] Installed student LAN isolation rules for 00-00-00-00-00-05
[firewall               ] Installed student LAN isolation rules for 00-00-00-00-00-06
[firewall               ] Installed student LAN isolation rules for 00-00-00-00-00-07
[firewall               ] [firewall] trigger OFF detected by ADMIN (S4)
[firewall               ] [firewall] Deleted all student LAN isolation rules
[learn_connections      ] [flow] Installing flow for 00-00-00-00-00-04.2 -> 00:00:00:00:00:01
[learn_connections      ] [flow] Installing flow for 00-00-00-00-00-04.3 -> 00:00:00:00:00:02
      ...
[learn_connections      ] [arp] Sending ARP reply from 10.0.0.6 to 10.0.0.7
[learn_connections      ] [flow] Installing flow for 00-00-00-00-00-07.1 -> 00:00:00:00:00:06
[firewall               ] [firewall] trigger ON detected by ADMIN (S4)
[firewall               ] Installed student LAN isolation rules for 00-00-00-00-00-05
[firewall               ] Installed student LAN isolation rules for 00-00-00-00-00-06
[firewall               ] Installed student LAN isolation rules for 00-00-00-00-00-07
     ...
```

(b) POX Controller output.

Figure 2: Mininet output showing connection states and table updates when turning firewall on (yellow) and off (blue).

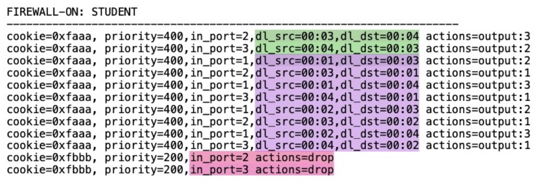


```
BOOTSTRAP: ADMIN
---------------------------------------------------------------------------
cookie=0xf0f0, priority=800, tcp, tp_dst=9999, actions=CONTROLLER:65535
cookie=0xf0f0, priority=800, tcp, tp_dst=8888, actions=CONTROLLER:65535
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:01, actions=output:2
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:02, actions=output:3
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:03, actions=output:1
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:04, actions=output:1
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:05, actions=output:1
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:06, actions=output:1
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:07, actions=output:1
cookie=0x0000, priority=50, dl_dst=00:00:00:00:00:08, actions=output:1
cookie=0x0000, priority=0 , actions=CONTROLLER:65535 # table miss

BOOTSTRAP: STUDENT
-------------------------------------------------------------------------
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:01 actions=output:1
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:03 actions=output:2
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:04 actions=output:3
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:02 actions=output:1
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:05 actions=output:1
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:06 actions=output:1
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:07 actions=output:1
cookie=0x0000, priority=50,  dl_dst=00:00:00:00:00:08 actions=output:1
cookie=0x0000, priority=0 , actions=CONTROLLER:65535 # table miss
```

Figure 3: Flow tables after bootstrap.

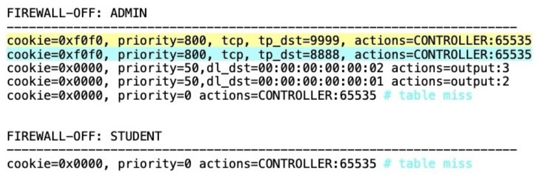
Figure 3 shows the state immediately after bootstrap. The ADMIN table contains the firewall trigger rules (highlighted in yellow and blue), and both switches have learned flows for reaching all eight hosts.



```
FIREWALL-ON: STUDENT
--------------------------------------------------------------------
cookie=0xfaaa, priority=400,in_port=2,dl_src=00:03,dl_dst=00:04 actions=output:3
cookie=0xfaaa, priority=400,in_port=3,dl_src=00:04,dl_dst=00:03 actions=output:2
cookie=0xfaaa, priority=400,in_port=1,dl_src=00:01,dl_dst=00:03 actions=output:2
cookie=0xfaaa, priority=400,in_port=2,dl_src=00:03,dl_dst=00:01 actions=output:1
cookie=0xfaaa, priority=400,in_port=1,dl_src=00:01,dl_dst=00:04 actions=output:3
cookie=0xfaaa, priority=400,in_port=3,dl_src=00:04,dl_dst=00:01 actions=output:1
cookie=0xfaaa, priority=400,in_port=1,dl_src=00:02,dl_dst=00:03 actions=output:2
cookie=0xfaaa, priority=400,in_port=2,dl_src=00:03,dl_dst=00:02 actions=output:1
cookie=0xfaaa, priority=400,in_port=1,dl_src=00:02,dl_dst=00:04 actions=output:3
cookie=0xfaaa, priority=400,in_port=3,dl_src=00:04,dl_dst=00:02 actions=output:1
cookie=0xfbbb, priority=200,in_port=2 actions=drop
cookie=0xfbbb, priority=200,in_port=3 actions=drop
```

Figure 4

Figure 4 shows the table updates after the firewall is enabled. High priority flow rules are installed for internal and admin traffic with lower priority rules for blocking everything else. The ADMIN table has no firewall rules installed.



```
FIREWALL-OFF: ADMIN
--------------------------------------------------------------------
cookie=0xf0f0, priority=800, tcp, tp_dst=9999, actions=CONTROLLER:65535
cookie=0xf0f0, priority=800, tcp, tp_dst=8888, actions=CONTROLLER:65535
cookie=0x0000, priority=50,dl_dst=00:00:00:00:00:02 actions=output:3
cookie=0x0000, priority=50,dl_dst=00:00:00:00:00:01 actions=output:2
cookie=0x0000, priority=0 actions=CONTROLLER:65535 # table miss

FIREWALL-OFF: STUDENT
--------------------------------------------------------------------
cookie=0x0000, priority=0 actions=CONTROLLER:65535 # table miss
```

Figure 5

Figure 5 shows the state after firewall deactivation. At this point, all flow entries are deleted and the controller forces the `ConnectionUp` routine. This re-installs the trigger rules on the ADMIN switch and the default table-miss rules across all switches.

When `pingAll` is run again, the tables return to the learned state of Figure 3a, and when the firewall is re-enabled, the Student rules from Figure 3b are reinstalled, further verifying the expected behaviour.

3. Wireshark Output

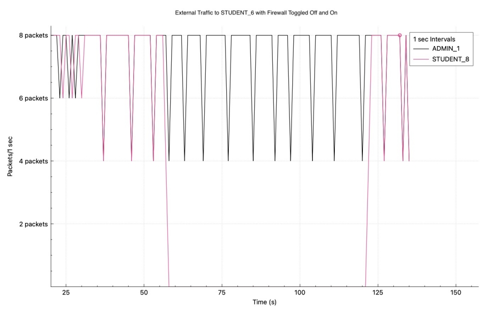
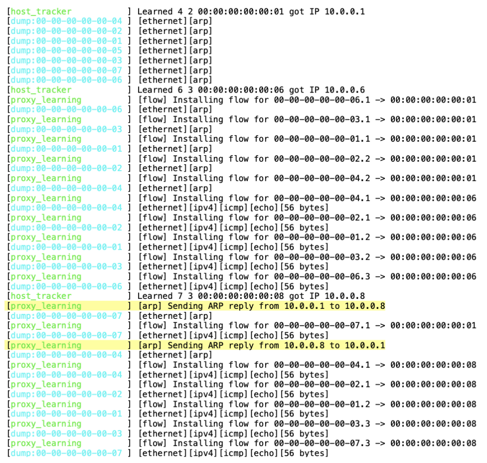


Figure 6

Figure 6 above shows the Wireshark I/O graph for traffic on `s6-eth3`, the interface to H6. H6 was pinged simultaneously from an ADMIN host (H1) and a Student host (H8). With the firewall disabled, both streams were visible at equal rates. When the firewall was enabled, traffic from H8 stopped completely while traffic from H1 continued. After disabling the firewall again, H8's traffic resumed. This confirms correct enforcement of the isolation rules.

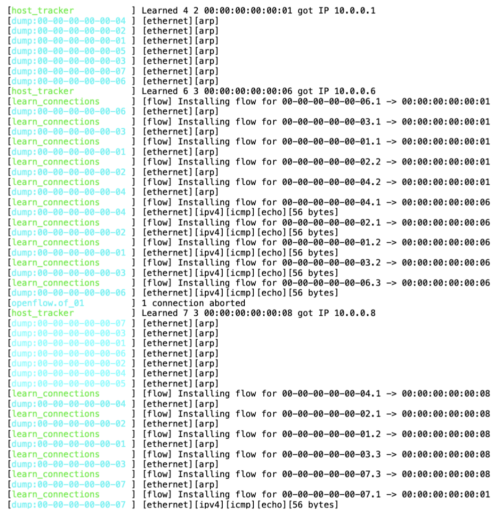
5.2 Proxy ARP Test Results

1. Controller Output



```
[host_tracker            ] Learned 4 2 00:00:00:00:00:01 got IP 10.0.0.1
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[dump:00-00-00-00-00-02 ] [ethernet][arp]
[dump:00-00-00-00-00-01 ] [ethernet][arp]
[dump:00-00-00-00-00-05 ] [ethernet][arp]
[dump:00-00-00-00-00-03 ] [ethernet][arp]
[dump:00-00-00-00-00-07 ] [ethernet][arp]
[dump:00-00-00-00-00-06 ] [ethernet][arp]
[host_tracker            ] Learned 6 3 00:00:00:00:00:06 got IP 10.0.0.6
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-06.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-06 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-03.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-03 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-01.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-01 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-02.2 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-02 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-04.2 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-04.1 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-04 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-02.1 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-02 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-01.2 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-01 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-03.2 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-03 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-06.3 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-06 ] [ethernet][ipv4][icmp][echo][56 bytes]
[host_tracker            ] Learned 7 3 00:00:00:00:00:08 got IP 10.0.0.8
[proxy_learning          ] [arp] Sending ARP reply from 10.0.0.1 to 10.0.0.8
[dump:00-00-00-00-00-07 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-07.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-07 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [arp] Sending ARP reply from 10.0.0.8 to 10.0.0.1
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-04.1 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-04 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-02.1 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-02 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-01.2 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-01 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-03.3 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-03 ] [ethernet][ipv4][icmp][echo][56 bytes]
[proxy_learning          ] [flow] Installing flow for 00-00-00-00-00-07.3 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-07 ] [ethernet][ipv4][icmp][echo][56 bytes]
```

Figure 7: Ping test with Proxy ARP



```
[host_tracker            ] Learned 4 2 00:00:00:00:00:01 got IP 10.0.0.1
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[dump:00-00-00-00-00-02 ] [ethernet][arp]
[dump:00-00-00-00-00-01 ] [ethernet][arp]
[dump:00-00-00-00-00-05 ] [ethernet][arp]
[dump:00-00-00-00-00-03 ] [ethernet][arp]
[dump:00-00-00-00-00-07 ] [ethernet][arp]
[dump:00-00-00-00-00-06 ] [ethernet][arp]
[host_tracker            ] Learned 6 3 00:00:00:00:00:06 got IP 10.0.0.6
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-06.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-06 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-03.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-03 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-01.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-01 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-02.2 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-02 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-04.2 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-04.1 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-04 ] [ethernet][ipv4][icmp][echo][56 bytes]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-02.1 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-02 ] [ethernet][ipv4][icmp][echo][56 bytes]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-01.2 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-01 ] [ethernet][ipv4][icmp][echo][56 bytes]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-03.2 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-03 ] [ethernet][ipv4][icmp][echo][56 bytes]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-06.3 -> 00:00:00:00:00:06
[dump:00-00-00-00-00-06 ] [ethernet][ipv4][icmp][echo][56 bytes]
[openflow.of_01          ] 1 connection aborted
[host_tracker            ] Learned 7 3 00:00:00:00:00:08 got IP 10.0.0.8
[dump:00-00-00-00-00-07 ] [ethernet][arp]
[dump:00-00-00-00-00-03 ] [ethernet][arp]
[dump:00-00-00-00-00-01 ] [ethernet][arp]
[dump:00-00-00-00-00-06 ] [ethernet][arp]
[dump:00-00-00-00-00-02 ] [ethernet][arp]
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[dump:00-00-00-00-00-05 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-04.1 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-04 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-02.1 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-02 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-01.2 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-01 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-03.3 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-03 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-07.3 -> 00:00:00:00:00:08
[dump:00-00-00-00-00-07 ] [ethernet][arp]
[learn_connections       ] [flow] Installing flow for 00-00-00-00-00-07.1 -> 00:00:00:00:00:01
[dump:00-00-00-00-00-07 ] [ethernet][ipv4][icmp][echo][56 bytes]
```

Figure 8: Ping test without Proxy ARP.

Figures 7 and 8 show the POX controller output for the first Proxy ARP test, generated by running `h1 ping -c1 h6` followed by `h8 ping -c1 h1`. Proxy

ARP maintains a database of learned MAC–IP mappings and suppresses broadcasts by generating synthetic ARP replies. In Figure 7, with Proxy ARP enabled, the controller responds directly to H8's request for H1, highlighted in yellow, avoiding a broadcast. In Figure 8, with Proxy ARP disabled, H8's ARP request is broadcast across the network before being resolved, confirming the expected suppression behaviour when Proxy ARP is active.

2. Wireshark Results

Figures 9 and 10 show startup traffic on the uplink of S2 (`s2-eth1`) during the bootstrap `pingAll`. With Proxy ARP enabled (9), requests and replies alternate in a steady, predictable pattern with relatively few broadcast spikes. Without Proxy ARP (10), the traffic is irregular and noisy, with bursts of overlapping requests and repeated broadcasts across the network.

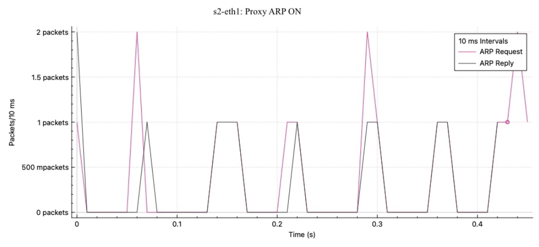


Figure 9

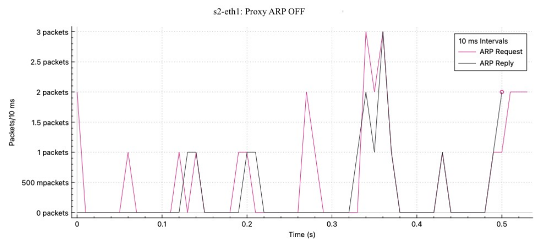


Figure 10

Figures 11 and 12 show captures taken across all S2 interfaces. With Proxy ARP enabled (11), requests and replies rise and fall together, converging quickly to a stable state. Without Proxy ARP (12), requests spike sharply and persist longer, as replies are slower to resolve and broadcasts continue to accumulate.

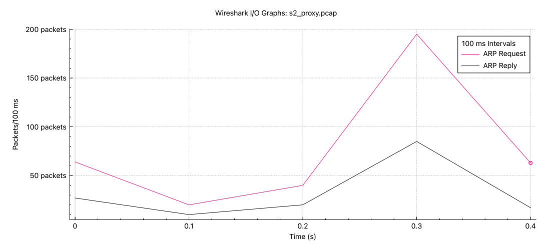


Figure 11

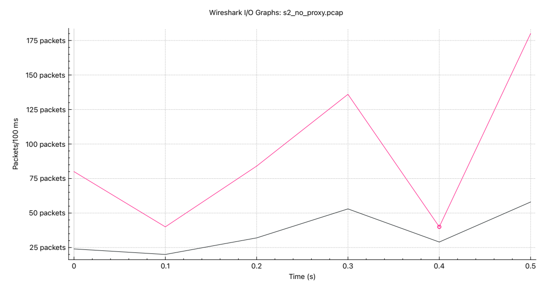


Figure 12

Numerical Results

Bootstrap traffic observed on `s2-eth1` with and without Proxy ARP:

Proxy ARP	**OFF**	**ON**	Δ%
Total Packets	44	30	-37.8%
Broadcasts (%)	63.6	60	-5.8%
Replies (%)	36.4	40	+9.4%
Time (s)	0.539	0.450	-17.9%

Bootstrap traffic (`pingAll()`) observed across all ports on S2 with and without Proxy ARP:

Proxy ARP	**OFF**	**ON**	Δ%
Total packets	776	541	-35.6%
Broadcasts (%)	72.2	70.6	-2.2%
Replies (%)	27.8	29.4	+5.6%
Time (s)	0.541	0.443	-19.9%

6 Discussion

This project demonstrated that SDN firewalls can be flexibly customized to enforce clearance-based isolation policies while remaining centrally managed. Although the test network was deliberately small, the principle scales: in more complex environments, such as campus or enterprise data centers, the ability to push firewall rules from a single controller simplifies administration compared to configuring each switch individually. The clear separation of ADMIN and Student LANs shows how SDN can enforce hierarchical clearance structures, a common requirement in government or research networks.

Further, the results showed that Proxy ARP reduced startup traffic significantly. On both uplink and aggregate captures, packet counts dropped by about 40% when Proxy ARP was enabled, and bootstrap completion time improved by roughly 20%. This reduction in broadcasts and faster convergence confirm the controller's effectiveness at suppressing ARP floods. While the testbed was small, the trend suggests that in larger topologies the savings would scale, making Proxy ARP a practical optimization for minimizing overhead in bootstrapping phases. This is conceptually similar to optimizations proposed by Dixit et al. [2], who maintain a "reachability map" during firewall initialization to avoid full propagation for every update. Both approaches highlight that initialization is a performance bottleneck and that caching or proxying mechanisms can mitigate repeated discovery costs.

Mininet provided a lightweight and controllable environment to prototype these ideas. While useful for experimentation, Mininet and POX are limited by their reliance on OpenFlow 1.0. Later OpenFlow versions and production controllers (e.g., Ryu, ONOS) provide richer match fields, cookie-based rule management, and more sophisticated policy abstractions [3], which would eliminate the need for coarse approaches like deleting all flows to disable the firewall. Still, the experiments here illustrate the underlying flexibility of controller-based policy enforcement.

7 Conclusion

Overall, the results confirm that centralized control enables not only policy customization but also performance optimizations that would be difficult with traditional distributed firewalls. The project illustrates how even simple controller logic can enforce clearance-based security policies while reducing unnecessary traffic.

A Appendix

A.1 Code

Source code used can be found here: Source code used can be found here: `https://github.com/ginanasseri/SDN-Firewall`

References

[1] Ahmadreza Montazerolghaem. Software-defined load-balanced data center: design, implementation and performance analysis. *Cluster Computing*, 24(2):591–610, 2021.

[2] Vikash Kumar Patidar and Abhishek Joshi. Challenges and preparedness of SDN-based firewalls. *arXiv preprint*, 2017. Accessed 2025-08-26.

[3] POX Documentation. Ofp_flow_mod — flow table modification (pox controlswitch example), n.d. Accessed 2025-08-26.