An Instructional Home Network Testbed for Undergraduate Networking Education

Andy Zheng[1,*], Adam Beauchaine[2], and Mira Yun[1]

[1]Department of Computer Science, Boston College, Chestnut Hill, MA, USA
[2]Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA, USA
Email: zhengno@bc.edu (A.Z.); ajbeauchaine@wpi.edu (A.B.); yunmd@bc.edu (M.Y.)
[*]Corresponding author
Manuscript received March 21, 2025; revised May 19, 2025; accepted July 4, 2025; published November 13, 2025

***Abstract*—Teaching undergraduates about home networking challenges is difficult, as applied networking concepts are often complex to demonstrate in a traditional classroom setting. Many existing simulation tools lack fidelity, transparency, or require expensive hardware and software. Moreover, industry-standard and research-grade tools tend to be overly complex for undergraduate students, making them hard to grasp and inefficient to teach. To address these challenges, this paper presents an instructional home network testbed built with affordable Raspberry Pi 4Bs, enabling hands-on and accessible learning. We introduce three learning activities designed to showcase the ability of this testbed: (1) leveraging Reinforcement Learning (RL) to dynamically optimize Smart Queue Management (SQM) settings, (2) applying supervised learning to classify network flows for traffic prioritization, and (3) simulating attack scenarios to analyze network vulnerabilities. By integrating Artificial Intelligence (AI) into networking education, this testbed provides a scalable and cost-effective platform for teaching a wide range of networking concepts. It offers students hands-on experience with real-world networking challenges, including working with machine learning models, conducting traffic analysis, and simulating attacks, all while utilizing simple, efficient, and cost-effective hardware.**

***Keywords*—educational testbed for undergraduate, AI-driven network optimization, learning activities for undergraduate, network security education**

I. Introduction

The COVID-19 pandemic has fundamentally reshaped how we learn and interact, driving a surge in internet traffic that persists today. As education, work, and socialization increasingly move online, critical weaknesses in the Internet's infrastructure have been exposed [1]. One issue that is often overlooked but widely experienced is bufferbloat, which refers to excessive latency and degraded network performance caused by misconfigured buffer settings in network devices. When packets arrive at a node, they are queued in its buffer. While buffers help manage congestion, they can become overwhelmed during high-traffic periods, significantly increasing network latency. This problem is particularly disruptive in latency-sensitive applications such as online gaming, video conferencing, and VoIP calls [2]. During the COVID-19 lockdowns, many home networks suffered from bufferbloat as households became increasingly dependent on digital communication.

Managing bufferbloat is challenging, especially for home network users. Techniques such as Smart Queue Management (SQM) require extensive knowledge of networking protocols [3], and existing solutions are often static, failing to adapt to dynamic network conditions [4]. Beyond home deployment, teaching undergraduates about home networking issues presents additional obstacles. Applied networking concepts can be difficult to demonstrate in a classroom setting, as many available simulations are low-fidelity, lack transparency, or require expensive software and hardware. Furthermore, industry and research tools are often too complex for undergraduates, making them difficult to grasp and time-consuming to teach. Accessing live networks or working with physical network setups involving multiple routers also introduces security risks and financial burdens.

To address these challenges, this paper presents an instructional home network testbed designed for students and educators to explore bufferbloat, Quality of Service (QoS), and security. We detail the construction of this testbed using affordable off-the-shelf hardware, specifically Raspberry Pi 4Bs, which offer greater flexibility and customization than traditional home networking devices. This hands-on approach allows students to apply fundamental networking concepts through direct experimentation.

Using this testbed, we propose three key learning activities. First, we develop a Reinforcement Learning (RL)-based environment to train a model that dynamically adjusts SQM settings. RL is particularly suited for optimizing complex problems beyond the scope of traditional machine learning algorithms. By exploring various parameter configurations and evaluating their performance, RL enables autonomous learning to minimize latency without compromising throughput. The second activity applies supervised learning to classify network flows based on Differentiated Services (DiffServ) labels. This classification technique helps identify traffic patterns, allowing students to prioritize flows and improve network performance under bufferbloat conditions. Lastly, we introduce a generic attack scenario to help students analyze and understand security vulnerabilities within the testbed. While not directly related to bufferbloat, this activity provides valuable insights into network security and reinforces the importance of securing home networks. By integrating Artificial Intelligence (AI) into networking education, this work provides a foundational approach for educators to enhance hands-on learning, bridging the gap between theoretical networking concepts and practical implementation.

II. Undergraduate Networking Education

Teaching networking to undergraduate students presents a persistent challenge due to the significant gap between technical complexity and students' prior experience. One

major difficulty lies in delivering complex theoretical content, such as diverse network protocols, roles, and layered architectures in an accessible and practical manner [5]. Traditional teaching methods often fall short in bridging this gap, proving inefficient at translating abstract concepts into real-world applications. Instructors must devote considerable time and resources to ensure students grasp foundational topics, often at the expense of deeper analysis or classroom engagement [6].

A further challenge is the limited opportunity to apply theoretical knowledge in practical, hands-on lab exercises. Physical testbeds that mirror industry-scale networks are frequently hindered by high costs, spatial demands, and hardware reliability issues [7]. While more affordable alternatives exist, many of these systems are not tailored to realistic small-scale environments, such as home or campus networks [8, 9].

Simulators offer an attractive alternative due to their cost-effectiveness, scalability, and risk-free nature. Cisco Packet Tracer, for example, is widely used and beginner-friendly, but suffers from low fidelity and limited extensibility. At the other end of the spectrum, ns-3 provides high realism but is overly complex for novice undergraduates and requires significant instructional investment that can detract from core course objectives. Furthermore, many simulation tools struggle to reflect current trends and innovations in networking.

To address these limitations, we propose a Raspberry Pi based testbed designed specifically for undergraduate education. The testbed is low-cost (approximately $100), accessible, and built on widely-used open-source software, making it both feasible and pedagogically effective. Its modular architecture supports a broad range of hands-on exercises in areas such as routing, network diagnostics, and performance analysis. The system is also easily customizable, enabling instructors to adapt lab content to varying course goals while exposing students to real-world tools and environments.

In addition, our testbed integrates emerging applications of AI into networking education, a domain that remains underrepresented in most undergraduate curricula. AI has shown great promise in areas such as network parameter optimization for wireless communication systems [10], traffic classification, and anomaly detection [11]. However, these innovations often rely on domain-specific datasets or computationally intensive models, making them difficult to implement in educational settings.

Our approach incorporates lightweight, modular AI components that are suitable for instructional use. This allows students to engage with AI-driven network tasks without requiring extensive infrastructure or advanced prerequisites. By doing so, we aim to both demystify AI applications in networking and better prepare students for emerging industry trends. Our testbed bridges the gap between affordability, simplicity, and relevance, making it a viable platform for modern, practice-oriented networking education.

III. Home Network Testbed: Design, Setup, and Configuration

To bridge theoretical concepts with practical implementation, this section introduces an affordable home network testbed designed for undergraduate students and educators. This testbed facilitates the exploration of various networking topics while integrating AI techniques into networking education. The following hardware and software components are required to build the instructional home network testbed.

1) Hardware
   - **Two Raspberry Pi 4B (RP4):** Serves as the access point and server.
   - **Micro SD Card:** At least 16 GB storage.
   - **Micro SD Card Reader:** To flash the firmware onto the SD card.
   - **Ethernet Cable:** For connecting the Raspberry Pi to a computer during initial setup.
   - **Laptop/PC:** For configuration, control, and training.
2) Software
   - **OpenWRT Firmware:** Download the appropriate version for RP4 from the OpenWRT website.
   - **Raspberry Pi Imager:** For flashing OpenWRT onto the SD card.
   - **Python Libraries:**
     - `Paramiko`: For Secure Shell (SSH) access.
     - `Ray Tune`: For RL-based tuning [12].
     - `PyTorch`: For implementing the RL model [13].
   - **Traffic Tools:**
     - `Netperf`: For traffic simulation and performance measurements [14].
     - `Flent`: For conducting Realtime Response Under Load (RRUL) tests [15].
     - `OpenScripts`: Used to fetch data about latency and network performance.

The required hardware and software components are all off-the-shelf, portable devices and open-source resources, ensuring accessibility and cost-effectiveness.

Our network topology, presented in Fig. 1, consists of four components: an access point running OpenWRT with the SQM package, a server, a traffic generator (acting as a transmitter-receiver pair), and a trainer for implementing RL algorithms. With one OpenWRT device serving as the main access point, all components can be wirelessly connected, creating a flexible and scalable testbed.

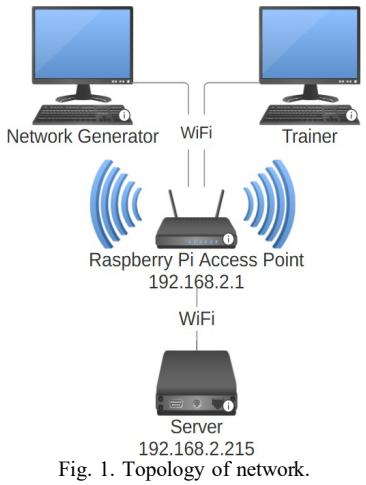

Fig. 1. Topology of network.

A. Preparing the Raspberry Pi with OpenWRT & SQM

Installing OpenWRT: To begin, download the appropriate

OpenWRT image file for the Raspberry Pi 4B from the OpenWRT website. Next, install Raspberry Pi Imager, which will be used to flash OpenWRT onto the microSD card. Once both files are prepared, insert the microSD card into the card reader and connect it to the laptop. Open Raspberry Pi Imager, select *Use Custom* under the *Operating System* option, and choose the *OpenWRT .img* file. Ensure the correct microSD card is selected under *Storage*, then write the OpenWRT image to the card.

After flashing OpenWRT, insert the microSD card into the RP4 and connect it to the laptop using an Ethernet cable. Open a web browser and navigate to 192.168.1.1, which will direct you to the *OpenWRT LuCI* interface. Since this is the initial setup, no password is required. Simply press *Enter* to proceed.

Setting up SQM: Next, we install and enable the SQM package. First, OpenWRT must be connected to the internet. Navigate to *Wireless* under the *Network* menu, then click *Scan* to detect available WiFi networks. Connect to the same WiFi network as the computer. A successful connection can be verified by monitoring the sent and received packet rates. RX and TX values are non-zero, indicating active data transfer between the Raspberry Pi and the WiFi network. Once connected to WiFi, navigate to the *System* tab and select *Software*. Click *Update* lists to run the *opkg package update* in the background. After the update completes, search for the *luci-app-sqm package* and click *Install*. The installation is successful when the screen resembles Fig. 2.

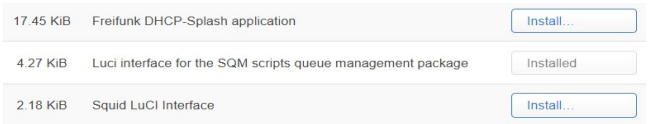

Fig. 2. A successful installation of the luci-app-sqm package.

B. Server and Traffic Generator Setup

To configure the server and traffic generator, we use the `Netperf` command line library. We can install `Netperf` on the second RP4 (server) and the traffic generator device. In our setup, we used a Macbook Pro as the traffic generator, but another RP4 can be used as a substitute. Install `Netperf` on both devices using the following command: `sudo apt update && sudo apt install netperf`. Next, run the appropriate command on each device:

- Server: `netserver`
- Traffic Generator: `netperf -H <server IP>-t TCP_RR -m <byte_size>`

To get the IP address of the server to fill in the `server_ip` field, run the command `hostname -I` on the server device. The `btye_size` field will be specified in the experimentation section.

IV. Implementing and Evaluating Reinforcement Learning in Network Performance

With the testbed set up, we can begin the first activity: integrating RL. This requires installing prerequisite libraries on the trainer device, which in our setup was a Lenovo Legion laptop. The primary RL library used is *Ray Tune*. To install Ray Tune, use the command: `pip install "ray[tune]"` [16]. The RL model follows a standard design with four core functions: initialize, reset, step, and update. These functions define how *Ray Tune* interacts with the environment during training. Below, we describe the implementation details of each function.

A. Initialization

Algorithm 1 Initialization Function
1: **Function** INITIALIZE():
2: Set environment specifications
3: Establish SSH connection to the server and configure network settings
4: Retrieve initial speed measurements
5: Define action and observation spaces
6: Initialize positions, speed, and maximum capacity
7: Reset environment

To begin, as shown in Algorithm 1, we can utilize the `paramiko` library to establish an SSH connection to the IP address `192.168.2.215`. This allows us to control one device from another device. Through this connection, we retrieve network speed measurements using the `betterspeedtest.sh` script from OpenScripts. Next, we initialize the action and observation space as required by *Ray Tune*. In our implementation, our action space is a 2-tuple of integers with possible integer values between -1 and 1. The observation space represents possible latencies, constrained to be in the range from 0 to our value defined by MAX_LAT. Finally, we initialize the network speed using *netperf* measurements and reset the environment to ensure a clean training state.

B. Reset

As shown in Algorithm 2, the reset function restores all variables to their initial state, ensuring a clean slate for *Ray* before the next iteration.

Algorithm 2 Reset Function
1: **Function** RESET(seed, options):
2: Reinitialize latency and system state
3: Reset reward and termination flag
4: Return the reinitialized state and info

C. Step

Algorithm 3 Step Function
1: **Function** STEP(action):
2: **if t**erminated **then**
3: Reset environment
4: **end if**
5: **if** count ⩾ MAX_STEPS **then**
6: Set terminated flag
7: **else**
8: Validate action
9: Increment step count
10: **if** position ⩾ MAX_LAT **then**
11: Set reward to REWARD_AWAY
12: **else**
13: Update speed based on action
14: **end if**
15: Retrieve current latency
16: Compute average latency
17: Set reward += REWARD(current latency, last latency, action)
18: Update state and info
19: **end if**
20: Update reward sum in info
21: Validate state
22: Return state, reward, terminated flag, False, info

The step function is where the core logic of the RL model is implemented. As shown in Algorithm 3, this function takes in a tuple `action` that tells *Ray* to either decrease (-1), increase (1), or keep (0) the current configurations. It takes the form of (ul_action, dl_action). For example, an action of (1,-1) tells ray to increase our upload limit, and decrease our download limit.

The step function first check the termination flag given by either a previous iteration of this step function (by surpassing the value set by MAX_STEPS), or by *Ray*. If the termination flag is set, the environment is reset for the next iteration. Otherwise, the action is validated against the predefined action space from the initialization phase, and the step count is incremented.

The next section of code is responsible for performing actions. If our current latency is higher than the maximum latency we allow as defined in our initialization, we punish the model by setting a negative reward, denoted as the constant `REWARD_AWAY`. If our latency is not greater than our maximum allowed latency, we call our update function, which will be described in the next section.

Finally, the function passes relevant information to the reward function, which evaluates performance by comparing the current latency with previous latencies, helping the model adjust its decision-making process.

Algorithm 4 Reward Function

```
1: Function REWARD(current latency, last latency, action):
2: if current latency ⩾ average latency + TOLERANCE then
3:     Set reward based on latency difference
4: else if current latency ⩾ last latency + TOLERANCE then
5:     Set reward based on latency difference
6: else
7:     Set reward to REWARD STEP and adjust based on action
8: end if
9: Return reward
```

As shown in Algorithm 4, the reward function supplements the step function by determining the reward for the most recent action. It evaluates performance by comparing the current latency with both the previous latency and the average latency, ensuring the model learns from past actions.

D. Update

Algorithm 5 Update Function

```
1: Function UPDATE(action):
2: Modify upload and download speeds based on action
3: Enforce maximum speed limits
4: Apply network configuration changes via SSH
5: Return updated speed settings
```

The update function executes the configuration changes requested by *Ray*. Using the `paramiko` library, we establish an SSH connection and apply the necessary updates to OpenWRT's SQM parameters. This is accomplished through a subprocess call that utilizes the native `uci set sqm` command. Once the changes are applied, the function returns the updated speed settings.

Additionally, several configurable constants allow students to modify parameters and observe their effects on model performance:

- MAX_LAT: Defines the maximum allowed measured latency before penalizing.
- MAX_STEPS: The number of iterations the step function runs before resetting
- REWARD_AWAY: The value of points to penalize unwanted behavior
- REWARD_STEP: The value of points to reward desired behavior
- TOLERANCE: Tolerance measures the amount of latency tolerated above our average and last latency measures, which is used when deciding to reward or penalize the model.
- HIST_COUNT: The number of previous latencies to factor into account when deciding to penalize or reward.

Encouraging students to experiment with these constants provides insights into how different configurations influence RL training and network performance.

E. Training

The primary experiment for students is to optimize OpenWRT parameters based on varying traffic conditions. To achieve this, we conduct three distinct trials using the traffic generator:

- **Empty Buffer:** No active traffic, representing a single-user scenario.
- **Moderate Traffic:** A 50 Mbps upload/download stream to mimic typical usage.
- **Heavy Traffic:** A 100 Mbps upload/download stream to simulate a highly stressed network.

The traffic load is specified by setting the `byte_size` field when running the traffic generator. Ensure that the traffic generator is actively transmitting data to the server before starting training.

Once traffic is being generated, we begin training *Ray* with the following parameters:

- **Algorithm:** Proximal Policy Optimization
- **Gamma:** 0.99
- **Learning Rate:** 0.01

Proximal policy optimization (PPO) is an actor-critic algorithm designed to constrain updates to the policy, which inherently makes it suitable for continuous action spaces. We use PPO because it is straightforward to implement and effectively regulates large parameter updates. This regulation is important, as substantial parameter changes can negatively impact performance if applied too aggressively.

We use a gamma (discount factor) of 0.99 to balance immediate rewards with the long-term improvements in latency. A learning rate of 0.01 provides enough parameter variability without causing excessive oscillations. These values were determined through trial and error across multiple performance tests. Students are greatly encouraged to experiment with these parameters to observe how different settings influence model convergence and final performance.

F. Evaluation

To evaluate the performance of our learned configurations, we use the *Flent* library. To install *Flent*, run the following command: `sudo apt install flent`. For ease of use, we utilize the graphical user interface (GUI) by running: `flent-gui`.

Next, we conduct Realtime Response Under Load (RRUL) tests. RRUL floods the network with packets, allowing users to observe bufferbloat under high traffic conditions. To initiate an RRUL test, navigate to *File* → *Run New Test* or

press *Ctrl* + *R*. The expected interface is shown in Fig. 3.

Once completed, we can use graphs to visualize our results. *Flent* includes various visualizations that each specialize in representing different metrics measured. For our purposes, *Flent* offers a box plot visualization that encapsulates our download, upload, and latency as shown in Fig. 4.

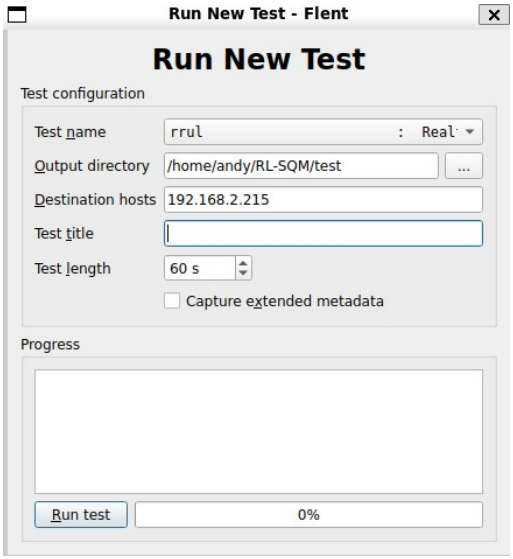

Fig. 3. Example of RRUL runtime.

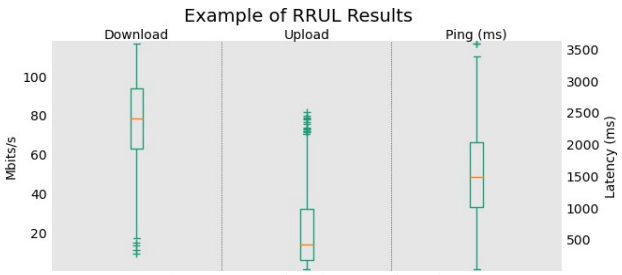

Fig. 4. Example of RRUL boxplot.

We run the RRUL test across all three configurations obtained from our model. The goal of this simulation is to evaluate how our learned configurations perform. For each network environment, we conduct RRUL tests under the following conditions: (1) No SQM enabled, (2) Default SQM configurations, (3) Learned SQM configurations. Students can observe the effects of SQM and analyze the trade-offs introduced by the learned model. They can use *Flent*-GUI's visual graphs to compare throughput and latency across different configurations.

V. Classifying Flows

An integral part of the SQM toolkit is a utility called DiffServ. DiffServ is a section of the IP header that assigns a priority level to each packet. SQM can read these headers to determine packet priority, ensuring efficient processing within a network node.

In an ideal scenario, DiffServ markers would accurately reflect packet priority based on network needs. However, in practice, this is often unreliable. Internet Service Providers (ISPs) may introduce bias by assigning higher priority to traffic from certain companies, manipulating DiffServ markings. Additionally, as packets traverse multiple network nodes, intermediate routers may modify or reset DiffServ bits, either ignoring them or applying their own rules. These inconsistencies hinder the reliability of DiffServ markings, potentially contributing to bufferbloat by delaying high-priority packets caught in the queue.

This activity attempts to demonstrate how to alleviate this problem when external packets enter a local network through the usage of classification. DiffServ employs several Differentiated Services Code Points (DSCPs), but only four major ones are commonly used: 0 (Default), 24 (CS2), 32 (CS4), and 46 (EF—Expedited Forwarding), listed in ascending priority

A. Creating the Dataset

In this experiment, we use the Unicauca Network Flows Dataset [17], but students are also encouraged to gather their own dataset using tools such as Wireshark. The Unicauca dataset includes the category of each flow instance, which we can map to known DSCP codes based on inferred ideas. For example, VoIP calls require low latency and should be assigned the highest priority (EF), while basic web browsing has minimal real-time requirements and is classified with the default priority (0). Students are encouraged to analyze the dataset and determine appropriate DSCP mappings based on their understanding of network traffic behavior.

B. Selecting Features

An essential step in this model is identifying the most relevant features for classification. From a user perspective, it is important to determine which packet attributes are useful in identifying traffic types and, consequently, their DSCP codes. For this experiment, we found the following features to be critical for classification: source IP, destination IP, source port, destination port, protocol, and packet size. Students are encouraged to explore additional features that may further enhance classification accuracy.

C. Training the Model

Once we have gathered the dataset and selected features, the next step is to train the classification model. For this experiment, we use a *random forest classifier*. However, students are encouraged to experiment with different models such as *support vector machines* or *neural networks*. We chose to use a *random forest classifier* for its robustness and resilience against overfitting. Additionally, students should fine-tune their models by adjusting hyper-parameters and conducting a grid search for optimization. For our *random forest classifier*, the optimal hyper-parameters were:

- max_depth: 100 (controls tree depth and manages complexity)
- criterion: log_loss (estimates the probabilities for each class)
- n_estimators: 50 (number of decision trees in the forest)
- min_samples_split: 2 (minimum samples required to split a node)
- class_weight: balanced_subsample (addresses class imbalances)

D. Results

To evaluate how well our model works, we split our dataset into parts and used a method called three-fold cross-validation. This means the model was trained on two-thirds of the data and tested on the remaining third, repeating this process three times with different data splits. We used settings, known as hyperparameters, that were

selected based on earlier testing. As shown in Fig. 5, the model performed very well, achieving approximately 96% accuracy and a macro F1-score of 0.92. These results show that the model is effective even when the number of examples in each category is uneven. For example, the CS4 category was the most difficult to classify correctly. EF packets made up only 3.7% of the data but accounted for just 2% of the total errors. The model was especially effective in distinguishing clearly different types of traffic, such as EF and Default DSCP markings, although its accuracy was slightly lower for intermediate categories like CS2 and CS4.

Accuracy: 0.9706917974006544

	precision	recall	f1-score	support
0	0.97	0.99	0.98	198008
24	0.96	0.85	0.90	8612
32	0.94	0.77	0.85	17355
46	0.98	0.95	0.97	8622
accuracy			0.97	232597
macro avg	0.97	0.89	0.92	232597
weighted avg	0.97	0.97	0.97	232597

Fig. 5. Classification report of best performing model.

This activity helps students understand how to classify network traffic using DSCP labels. It also encourages them to explore why some traffic is misclassified and consider potential sources of error or bias. Through this process, students gain practical experience in building more accurate models and improving how traffic is managed in real-world networks.

Similar activities can be applied to security label classification. Students with basic networking knowledge can learn to identify the security level of traffic based on network features, similar to techniques used in modern firewalls and data loss prevention (DLP) systems. Our testbed could be expanded to include dedicated nodes that generate malicious traffic using open-source security tools, allowing students to analyze and classify attack patterns in a safe environment. Deep packet inspection (DPI) could also be introduced to help students examine application-layer details in the traffic. These experiments can use supervised learning techniques, as demonstrated in this study, or explore unsupervised learning when labeled data is not available. Future work should continue developing these applications to support network security education and practical classification methods.

VI. Simulating and Analyzing Attacks on the Testbed

Using our previous tools, we can also perform an attack simulation on our network setup and evaluate the results. We can again utilize *Flent*'s RRUL test to simulate a Denial of Service (DoS) attack. This allows students to analyze the effects of such attacks and gain a deeper understanding of the vulnerabilities within our testbed.

Students can simulate this attack under two different conditions:

- Conducting the attack throughout the full length of training
- Conducting the attack for only a portion of training

For this experiment, we must modify our network configuration. Specifically, we reconfigure the traffic generator to act as an attacker by executing the RRUL test. Once this topology is set up, we can initiate the attack and observe its impact on network performance.

A. Full-Length Attack Simulation

In our first attack scenario, we examine the effects of a sustained attack conducted throughout the entire training duration. To execute this attack, we initiate an RRUL stress test using *Flent*. While the *Flent* GUI allows tests of up to three hours, longer stress tests must be performed using the command line. To launch an RRUL-based DoS attack via the command line, use the following syntax:

```
flent -H <Host IP> -L <Duration in Seconds> rrul
```

For our testbed, we will simulate an attack lasting 8 hours. This can be executed with the following command:

```
flent -H 192.168.2.215 -L 28800 rrul
```

This prolonged attack enables students to observe its impact on network performance over an extended period and evaluate how SQM and RL-based optimizations respond under persistent adversarial conditions.

Notably, in several instances, the training process crashed due to excessive congestion. This occurred because the SSH packets required by the model were dropped when the buffer became full, preventing successful initialization. However, in some cases, the model was still able to establish an SSH connection to OpenWRT and continue training.

As shown in Fig. 6, the DoS attack significantly increased network latency, fluctuating between 1 to 10 seconds of delay. Interestingly, despite the high latency, throughput remained stable and steadily improved over time. This suggests that while latency was severely impacted, packet delivery rates were not fully disrupted under these conditions.

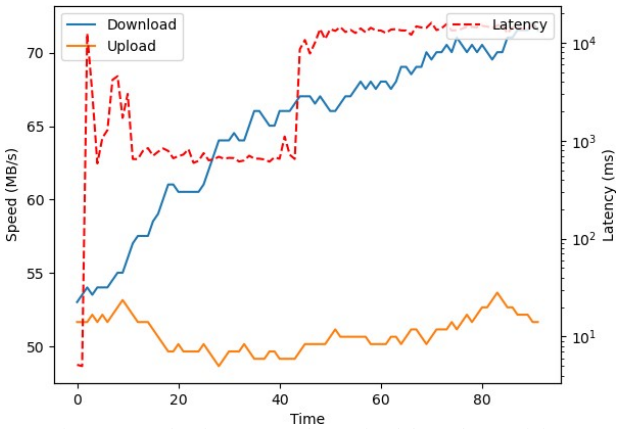

Fig. 6. Example of a DoS attack sustained throughout training.

B. Partial Attack During Training

Another attack scenario involves targeting the training process while it is in progress. In this example, we simulate the effects of a DoS attack occurring at specific intervals during model training.

In Fig. 7, we observe the direct impact of the attack, which ultimately caused the training process to crash. The latency spike at the tail end of the training session suggests that excessive congestion overwhelmed the system, preventing further progress.

Conversely, in Fig. 8, we performed the same attack, but

this time, the training process did not crash. Instead, we observe significant latency spikes during the attack, yet the model continued to train. Notably, despite the attack, major throughput improvements were still observed, particularly in download performance.

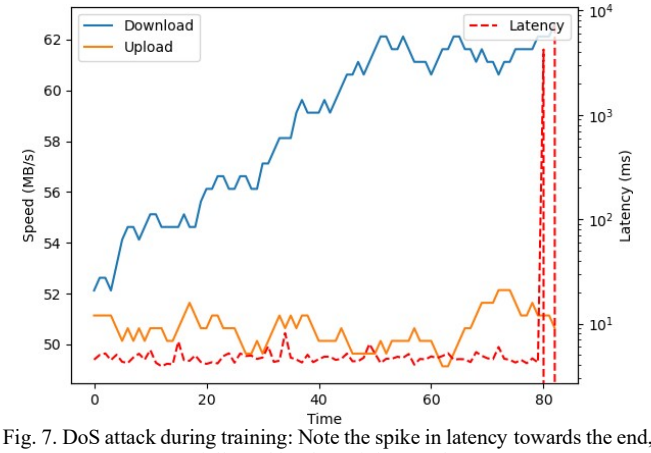

Fig. 7. DoS attack during training: Note the spike in latency towards the end, followed by the training crash.

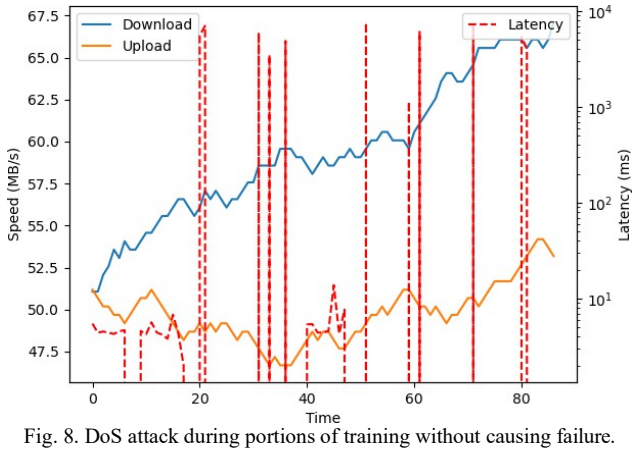

Fig. 8. DoS attack during portions of training without causing failure.

C. Analyzing the Impact of Attacks on Training

Across all tests, we observe a consistent trend: download throughput steadily increases, whereas upload throughput remains relatively constant. This suggests that the download interface of our system was initially affected by bufferbloat but gradually improved because of model training. Aside from instances where the attack caused training to crash, the training process itself showed no significant reduction in terms of learning effectiveness. However, one notable consequence of the attack was its effect on training time. Due to the high latency introduced by the attack, each training iteration took significantly longer to complete. Thus, while our model was still able to increase throughput despite latency spikes, the overall time required to reach an optimal configuration was negatively impacted.

VII. Conclusion

In this study, we developed an instructional home network testbed to provide undergraduate students with a hands-on learning experience in networking concepts. The testbed was built using affordable Raspberry Pi 4Bs and open-source software, making it an accessible and scalable tool for networking education. Through a series of learning activities, students explored the impact of bufferbloat, utilized RL to optimize SQM settings, and applied supervised learning to classify network flows based on DSCPs. Additionally, attack simulations using *Flent*'s RRUL test allowed students to analyze network vulnerabilities, including the effects of DoS attacks on training performance.

This work lays the groundwork for integrating AI techniques into networking education by providing students with practical experience in applying machine learning to network performance optimization and security analysis. The instructional materials were developed based on fundamental networking knowledge, making them accessible to undergraduate learners. The proposed activities were designed, implemented, and refined through the process of building and testing the testbed itself. We plan to incorporate these materials into a new undergraduate Computer Science elective at Boston College in the near future. This implementation will enable us to collect student feedback and assess the testbed's impact and effectiveness in a real classroom environment.

Future work may extend this testbed to incorporate unsupervised learning approaches and additional security-focused experiments, further broaden its educational scope and real-world applicability. By bridging theoretical networking concepts with practical implementation, this testbed provides an engaging, scalable, and cost-effective solution for modernizing undergraduate networking curricula.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

AZ proposed the research idea, designed and conducted the experiments, performed the testing, and wrote the manuscript. AB collaborated closely with AZ throughout the project, contributing to design discussions, implementation, and refinement. MY supervised the project, provided guidance and feedback at all stages, reviewed the manuscript, and approved the final version. All authors read and approved the final manuscript.

References

[1] R. Ranganathan, C. Markovich, T. Mathur, O. Abu-Hijleh, T. Cloonan, and J. Ulm, "Decoding the bandwidth surge during COVID-19 pandemic," *NCTA Technical Papers*, 2020.

[2] J. Gettys, "Bufferbloat: Dark buffers in the internet," *IEEE Internet Computing*, vol. 15, no. 3, pp. 96–96, 2011.

[3] A. Zheng and M. Yun, "Affordable self-tuning smart queue management for home networks," in *Proc. 2024 IEEE 15th Annual Ubiquitous Computing, Electronics Mobile Communication Conference (UEMCON)*, 2024, pp. 271–277.

[4] J. Ye, K. Cai, D. Lin, J. Li, J. He, and J. C. Lui, "A control-theoretic and online learning approach to self-tuning queue management," in *Proc. 2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, 2022, pp. 1–10.

[5] R. Sbeit and P. Wang, *Using a Team Project in Teaching Computer Networking and Data Communication*, 2021.

[6] S. Zhou *et al*., "The construction and practice of computer network course driven by innovative thinking," *MATEC Web Conf.*, vol. 309, 05017, 2020. https://doi.org/10.1051/matecconf/202030905017

[7] E. J. Rodr'ıguez-Seda, P. J. Frontera, and J. Bradshaw, "A networked cyber-physical system testbed for undergraduate education," in *Proc. IECON 2018 - 44th Annual Conference of the IEEE Industrial Electronics Society*, 2018, pp. 3007–3012.

[8] C. W. Sørensen *et al*., "Easy as pi: A network coding raspberry pi testbed," *Electronics*, vol. 5, no. 4, 2016.

[9] S. Laki *et al*., "P4pi: P4 on raspberry pi for networking education," *SIGCOMM Comput. Commun. Rev.*, vol. 51, no. 3, pp. 17–21, Jul. 2021. https://doi.org/10.1145/3477482.3477486
[10] S. Vangaru *et al.*, "A multi-agent reinforcement learning testbed for cognitive radio applications," arXiv preprint arXiv:2410.21521, 2024.
[11] R. Kozlica *et al.*, "A modular test bed for reinforcement learning incorporation into industrial applications," arXiv preprint arXiv:2306.01440, 2023.
[12] Z. Wu, E. Liang, M. Luo, S. Mika, J. E. Gonzalez, and I. Stoica, "RLlib flow: Distributed reinforcement learning is a dataflow problem," in *Proc. Conference on Neural Information Processing Systems (NeurIPS)*, 2021.
[13] A. Paszke *et al*., "Pytorch: An imperative style, high-performance deep learning library," *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019, pp. 8024–8035.
[14] Netperf. (2021). [Online]. Available: https://github.com/HewlettPackard/netperf
[15] T. Høiland-Jørgensen, C. A. Grazia, P. Hurtig, and A. Brunstrom, "Flent: The flexible network tester," in *Proc. the 11th EAI International Conference on Performance Evaluation Methodologies and Tools*, New York, NY, USA: Association for Computing Machinery, 2017, pp. 120–125. https://doi.org/10.1145/3150928.3150957
[16] E. Liang *et al*., "RLlib: Abstractions for distributed reinforcement learning," presented at International Conference on Machine Learning (ICML), 2018.
[17] J. S. Rojas, A. Pekar, Rendo´n, and J. C. Corrales, "Smart user consumption profiling: Incremental learning-based ott service degradation," *IEEE Access*, vol. 8, pp. 207 426–207 442, 2020.