CHAPTER 5
CODING AND TESTING

This chapter describes the AWS and Machine Learning-based DDoS Protection System implementation using code, along with testing approaches used to validate that the system works as anticipated. The implementation was about creating synthetic traffic data to generate anomalies using the Isolation Forest algorithm and proving scalability and resilience of the system in DDoS situations. Testing includes load testing with Apache JMeter, performance monitoring with AWS CloudWatch, and validation of the system's ability to withstand DDoS attacks while maintaining availability.

5.1 Traffic Data Generation

To simulate real-world web traffic, a Python script was developed to generate synthetic traffic data. This data is both normal web traffic and DDoS-like malicious traffic. The synthetic traffic data is created using the Poisson distribution with random anomalies at intervals that reflect sudden surges in traffic common in DDoS attacks.

5.1.1 Pseudocode

```
BEGIN
    # Set the random seed for reproducibility
    SET random seed to 0
    # Define the number of data points (1440 for one day's worth of data)
    SET data_points to 1440
    # Generate traffic data using Poisson distribution with a mean of 50 requests per
minute
    GENERATE traffic data using Poisson distribution with mean 50 for data_points
    # Introduce anomalies by selecting random indices and multiplying their traffic
value by 5
    SELECT 10 random indices from traffic data points
    FOR each selected index DO
        MULTIPLY the traffic value at that index by 5
    END FOR
    # Create a DataFrame with the generated traffic data
    CREATE a DataFrame with 'Traffic' column containing the traffic data
    # Save the generated traffic data to a CSV file
    SAVE the DataFrame to a file named 'traffic-data.csv' without index
    # Print a success message
    PRINT "Synthetic traffic data saved to 'traffic-data.csv'"
END
```

5.1.2 Testing the Traffic Data Generation

Generated synthetic traffic data is stored as a CSV file and utilized in feeding the anomaly detection model. Artificial anomalies are generated by multiplying random data points with a factor of five that reflects the typical spiky patterns found in DDoS attacks. Traffic data was also saved and processed to ensure ready usage in further testing processes.

5.2 Dataset Generation

For anomaly detection, the algorithm of choice was Isolation Forest from the scikit-learn library. It uses recursive partitioning of the data to isolate anomalies, separating them from most data points. The process begins by loading synthetic traffic data generated from above, followed by the training of the Isolation Forest model to classify traffic as either normal or anomalous.

5.2.1 Anomaly Detection with Isolation Forest

For the simulation, traffic was captured by logging every packet passed to the router. This included benign and malicious traffic. The data logged in an XML file meant that through that format, I was able to capture critical details such as timestamp, source IP, destination IP, and packet size. Pseudocode is as follows:

```
BEGIN
    # Load the traffic data from a CSV file
    LOAD traffic data from 'traffic-data.csv' into a DataFrame
    # Display the first few rows of the loaded data
    PRINT the first few rows of the data
    # Extract the 'Traffic' column and store it in variable X
    SET X to the 'Traffic' column values from the data
    # Extract the true labels (normal or attack) from the 'True_Label' column
    SET y_true to the 'True_Label' column values from the data
    # Initialize the Isolation Forest model with contamination level of 0.05
    INITIALIZE IsolationForest model with contamination set to 0.05
    # Fit the model to the traffic data (X)
    FIT the IsolationForest model on X
    # Predict anomalies using the model (-1 for anomalies, 1 for normal)
    PREDICT anomalies using the IsolationForest model on X
    # Add the predictions to the DataFrame as a new 'Anomaly' column
    ADD 'Anomaly' column to the data with the predicted anomalies
    # Filter and store the rows where anomalies are detected (Anomaly = -1)
    SET anomalies to the rows where 'Anomaly' equals -1
    # Print the detected anomalies
    PRINT the anomalies
    # Calculate the accuracy by comparing predicted anomalies with true labels
    CALCULATE accuracy using accuracy_score(y_true, predicted anomalies)
```

```
      # Print the accuracy percentage
      PRINT the accuracy as percentage
      # Generate confusion matrix
      CALCULATE confusion matrix using confusion_matrix(y_true, predicted anomalies)
      # Print the confusion matrix
      PRINT the confusion matrix
      # Generate classification report for precision, recall, and F1 score
      PRINT classification report using classification_report(y_true, predicted
anomalies)
      # Plot the traffic data and highlight anomalies
      CREATE a plot with traffic data
      PLOT traffic data as a line graph
      PLOT anomalies as red scatter points on the same graph
      LABEL x-axis as 'Time' and y-axis as 'Traffic'
      ADD legend to the plot
      # Save the plot to a file
      SAVE the plot as 'anomalies_plot.png'
  END
```

5.2.2 Testing Anomaly Detection

The model has been trained on synthetic data, which is the traffic information. Then, anomalies are determined by patterns in the traffic. Its performance was compared by metrics such as accuracy, precision, recall, and F1 score. A confusion matrix was formed to compare the predicted outcomes with true labels (normal vs. DDoS). Detected anomalies were plotted to visualize the irregular traffic spikes.

5.3 Load Testing with Apache JMeter

Apache JMeter is used for testing load against heavy traffic loads and DDoS attacks for validating the resilience of the system. Tests include various scenarios for different concurrent numbers of users, ramp-up periods, and loop counts that might represent a DDoS attack condition in the real world to analyze how well the system performs when a traffic load comes unexpectedly.

Apache JMeter was set with a multiple user simulation, using HTTP requests to the server. It was run in stress modes that varied parameters like user numbers, ramp-up times, and loop counts as stress levels on the system for responses during varied traffic conditions. That would mean observing the server regarding how it would respond regarding handling high traffic and available service.

5.4 Cloud Integration with AWS

The anomaly detection and traffic monitoring components of the system were deployed on AWS EC2 instances. This cloud setup allowed for real-time monitoring and automatic scaling based on incoming traffic. NetworkIn (incoming network traffic) and CPU utilization were monitored using Amazon CloudWatch. Alerts were configured to notify administrators if certain thresholds were breached, providing real-time performance insights.

5.4.1 AWS CloudWatch Setup

Command to run:

```
aws cloudwatch put-metric-alarm --alarm-name "HighTrafficAlarm" --metric-name NetworkIn --namespace AWS/EC2 --statistic Sum --period 60 --threshold 1000000 --comparison-operator GreaterThanThreshold --dimensions Name=InstanceId,Value=i-1234567890abcdef0 --evaluation-periods 1 --alarm-actions arn:aws:sns:region:account-id:alarm-action
```

5.4.2 Testing AWS CloudWatch Alerts

CloudWatch metrics were monitored in real-time while doing load testing. Alerts were set up to notify the administrators if traffic thresholds were exceeded. This way, swift action could be taken in case of a possible DDoS attack, thus keeping the system responsive and available even during stress conditions.

5.5 Auto Scaling and Load Balancing

The system utilizes AWS Auto Scaling, which dynamically changes the number of EC2 instances according to the level of traffic load. During increased traffic volume, Auto Scaling adds more instances to handle the load, and with decreased traffic, it scales down the number of instances. This is done through the AWS Elastic Load Balancer, which distributes the incoming traffic evenly across instances, thus not overwhelming any particular server and ensuring service availability.

5.5.1 Testing Auto Scaling and Load Balancing

Command to run:

```
aws autoscaling create-auto-scaling-group --auto-scaling-group-name MyAutoScalingGroup --launch-configuration-name MyLaunchConfig --min-size 1 --max-size 10 --desired-capacity 2 --vpc-zone-identifier subnet-abc123
```

5.5.2 Testing Load Balancer Behavior

Testing the performance of the load balancer was done through simulation of traffic spikes. In this, traffic distribution between the EC2 instances was observed to prevent overloading of any one instance due to excessive traffic and to ensure high availability of the system.

5.6 System Validation and Performance Evaluation

Several attack scenarios were simulated using Apache JMeter to validate the system. The performance of the anomaly detection system, Auto Scaling, and Load Balancer in ensuring service availability against DDoS attacks was evaluated. Detection accuracy, scaling efficiency, and system availability were some of the performance metrics analyzed.

Performance metrics from the tests were used to evaluate:

- Detection accuracy: The system's ability to correctly identify anomalous traffic.
- Scalability: The capability of the system to bear very high traffic volumes, made possible by AWS Auto Scaling.
- Availability: The uptime percentage in simulated DDoS attack conditions.

Results obtained through these tests provided much-needed insights into the strengths and weaknesses of the system's anomaly detection and mitigation, affirming the efficiency of the DDoS protection system.