A Deep Learning based Cloud Intrusion Detection System

Video Demo & Explanation: B22CS011_B22CS026_VCC_Project.mp4

Table of Contents

Problem Statement

Intrusion detection is one of the technologies that protects the cloud computing environment from malicious attacks. However, network traffic in the cloud computing environment is characterised by large scale, high

dimensionality, and high redundancy, and these characteristics pose serious challenges to the development of cloud intrusion detection systems (IDS). Deep learning (DL) techniques have shown considerable potential for intrusion detection. We want to devise and test the effectiveness of various DL models for intrusion detection in a real-world scenario through a self-created application hosted across cloud virtual machines (VMs).

Literature Reference & Existing Results

IDS for cloud environments have become increasingly complex due to the growing volume and complexity of cyber threats. Traditional signature-based IDS approaches often fail to detect such novel attacks. High-dimensional and large-scale network traffic associated with the cloud computing environment makes things more complicated. Recent research has turned towards DL-based methodologies for better anomaly detection.

Wang et al. (2022) proposed a hybrid intrusion detection framework that leverages a Stacked Contractive Autoencoder (SCAE) for unsupervised feature extraction and then utilise a Support Vector Machine (SVM) as the final layer for the classification task.

. The SCAE tries to reduce the high-dimensional network data into a more compact representation that is more informative for performing the classification. They benchmarked their model evaluations on NSL-KDD and KDD Cup 99 datasets, with a high accuracy of 0.92 and 0.99 in the respective datasets.

Building upon this, Dalal et al. (2023) introduced an Extremely Boosted Neural Network to predict multi-stage cyberattacks in a cloud computing context. They wanted to train their model on the Multi-Step Cyber-Attack Dataset (MSCAD) to create a model that is stage-aware, thus better suited for advanced persistent threats (APTs) and zero-day attacks. Their model consisted of an ensemble of neural networks, which was able to achieve a remarkable 99.72% detection accuracy on the MSCAD dataset, surpassing conventional approaches such as Bayesian Networks and QUEST decision trees.

Inspired by this, we decided to utilise a DL regularisation technique known as dropout, which we studied to have similar performance as ensemble learning but with a fraction of the network size.

Implementation

After observing these promising results in these studies, we developed the DL-based IDS using a deep neural network architecture. The model consisted of 3 hidden MLP layers, with dropout with p=0.3 applied on the first two hidden layers. A general logic in designing neural networks is to have the first hidden layer size to be greater than the input size (=31). Thus, the hidden layer size was chosen to be 64.

The model was trained and validated using the KDD Cup 99 dataset. The model exhibited a very high train accuracy of 0.9994 & validation accuracy of 0.9996 on the 5-class classification task.

Given the encouraging performance, we decided to move further with the model and test it in a real-world setting by testing it on network packets captured from a Google Cloud Platform (GCP) VM hosting an exploitable web application that we designed and deployed. This deployment allowed us to evaluate the model's reliability under realistic network conditions, with incoming traffic logged and processed continuously.

Step-by-step implementation

Creating the model

The Python Pytorch library was used to build the model, and the dataset was sourced from Kaggle. After sufficient data pre-processing, the model training was set for 20 epochs with early stopping in case validation accuracy decreases.

Deploying the model

Once the model was trained, it was deployed on a GCP VM so that it could be queried by the system running the application for prediction. The VM on which the model was deployed was separate from the VM running the app, since the DL model may be resource-intensive and thus may hinder application performance.

Creating the deployment code

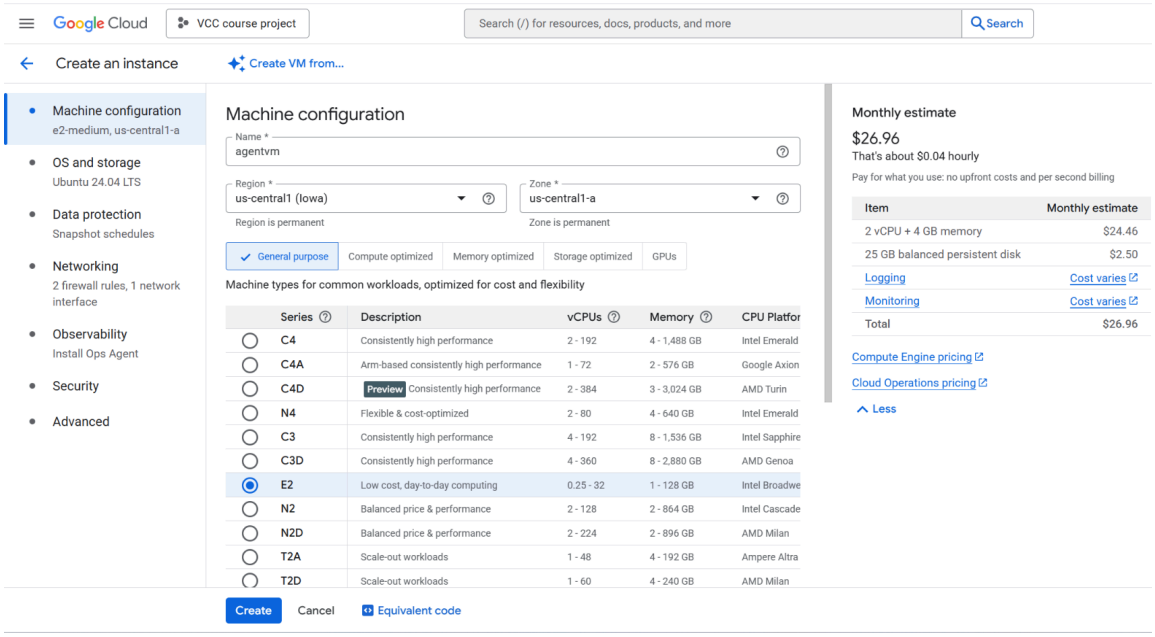
After training, the trained model was saved as a .pt file. We wrote a Python script, 'agent.py', that listens (using socket programming) for queries to get the prediction for the data sent over the network.

It then uses the saved DL model to get the prediction(s) and sends them back to the client.

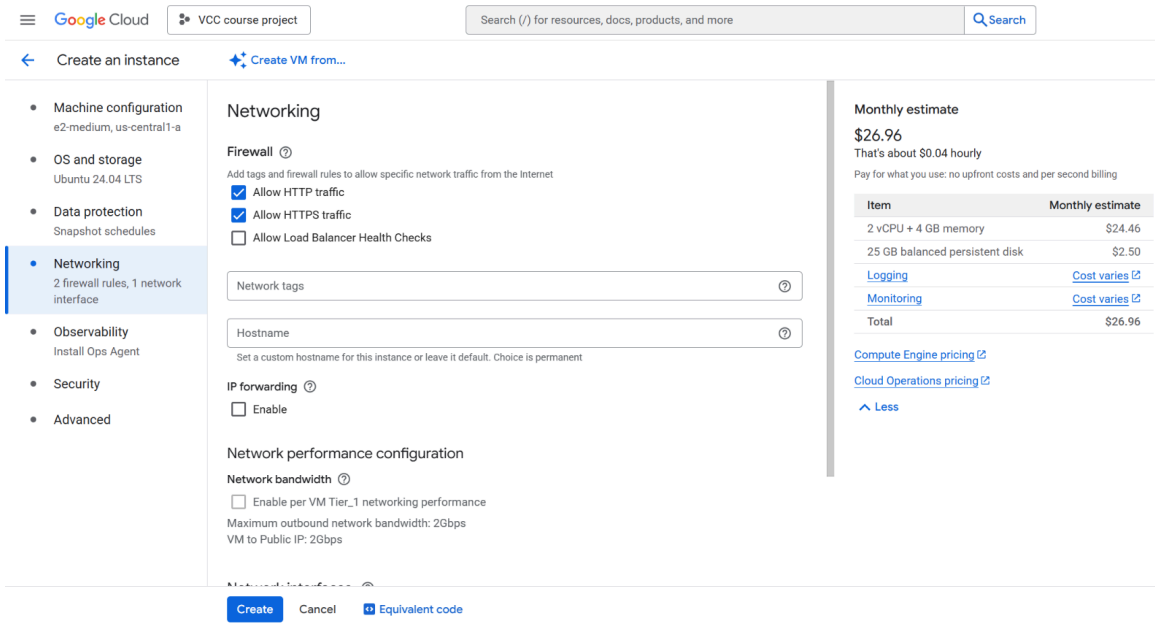
Creating the AI Agent VM

A GCP VM with the following configuration was created-

1. Region: us-central1-a
2. Machine Type: e2-medium (2 vCPU, 1 Core, 4GB memory)
3. OS: Ubuntu 24.04 LTS
4. Storage: 25 GB balanced persistent disk (more than the standard 10GB since the 'flask' python library is very large)
5. Firewall: allow HTTP & HTTPS traffic



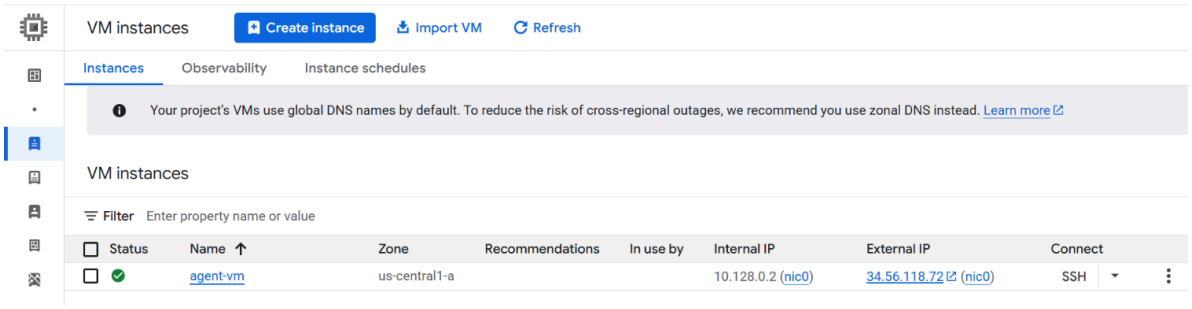
Creating the agent VM with the above configurations



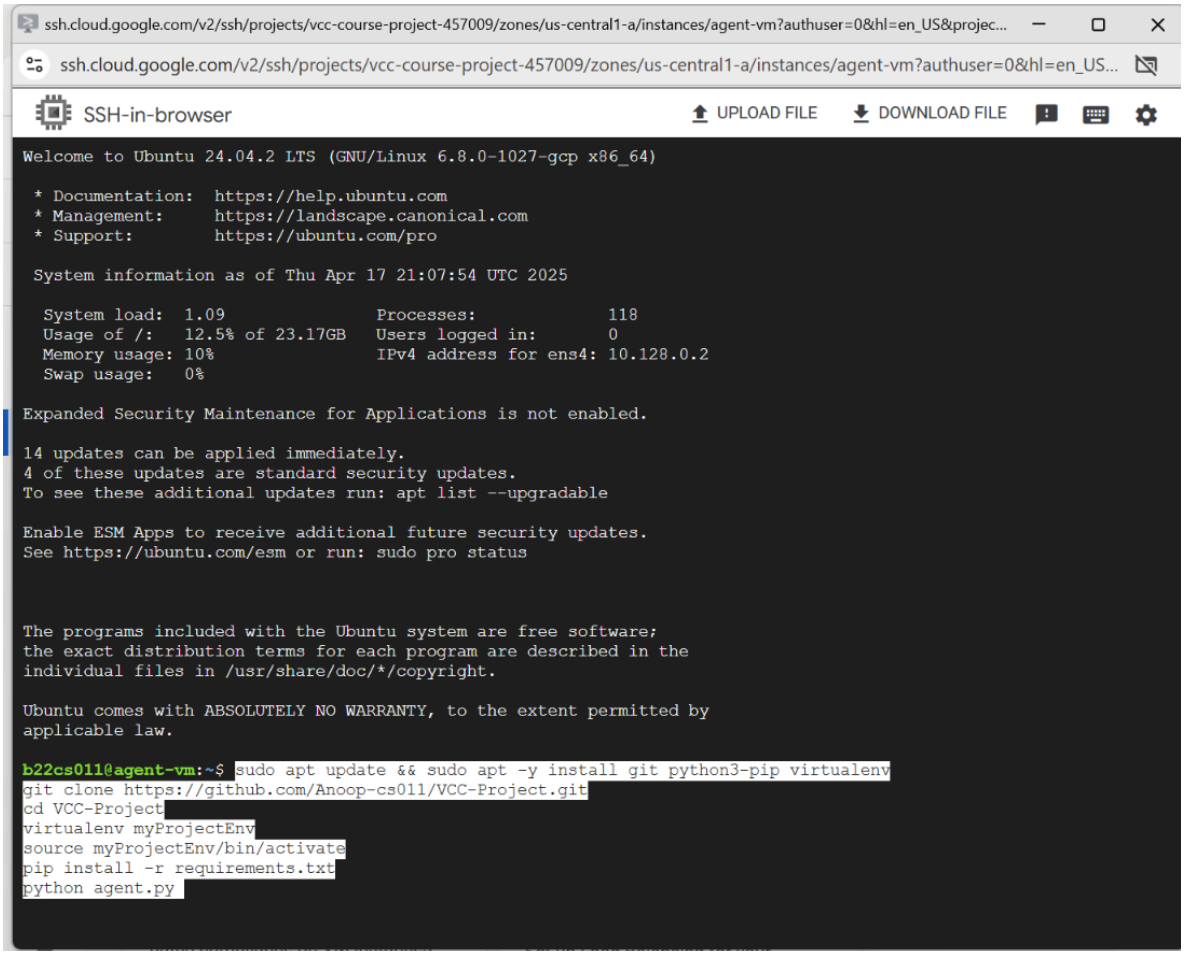
Changing the network settings to allow internet communication between the agent VM and the app VM

Running the startup script

The following shell script was executed to install the required libraries and run the “AI agent” script. This script can be found in the project’s git repository.



Start the agent VM and connect to the secure shell (SSH)



Copy and execute the agent's shell script.

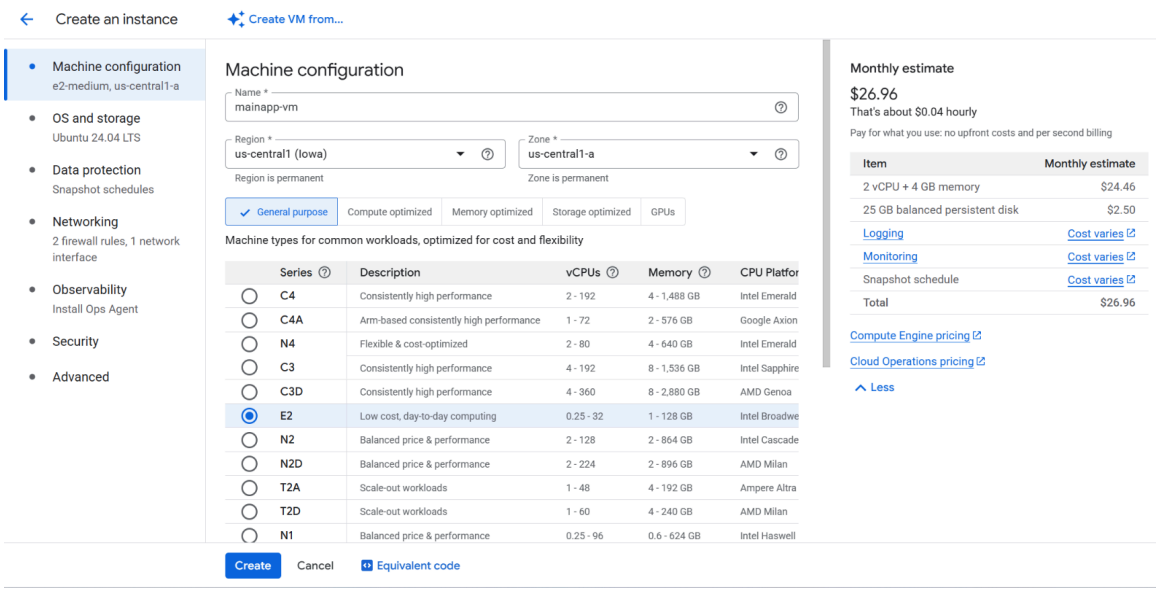
Once the agent script is loaded, it will be available for prediction

Creating and deploying the exploitable web application

Creating the app hosting VM

A GCP VM with the following configuration was created-

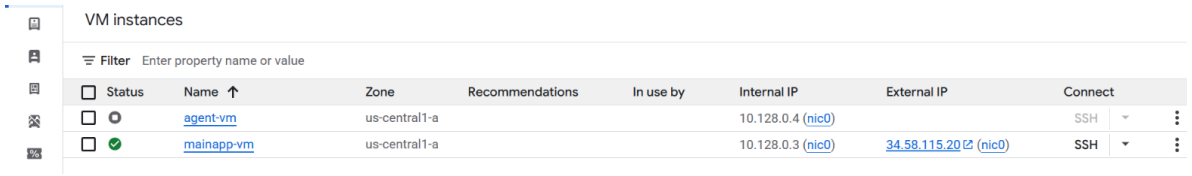
1. Region: us-central1-a
2. Machine Type: e2-medium (2 vCPU, 1 Core, 4GB memory)
3. OS: Ubuntu 24.04 LTS
4. Storage: 25 GB balanced persistent disk (more than the standard 10GB since the ‘flask’ python library is very large)
5. Firewall: allow HTTP & HTTPS traffic



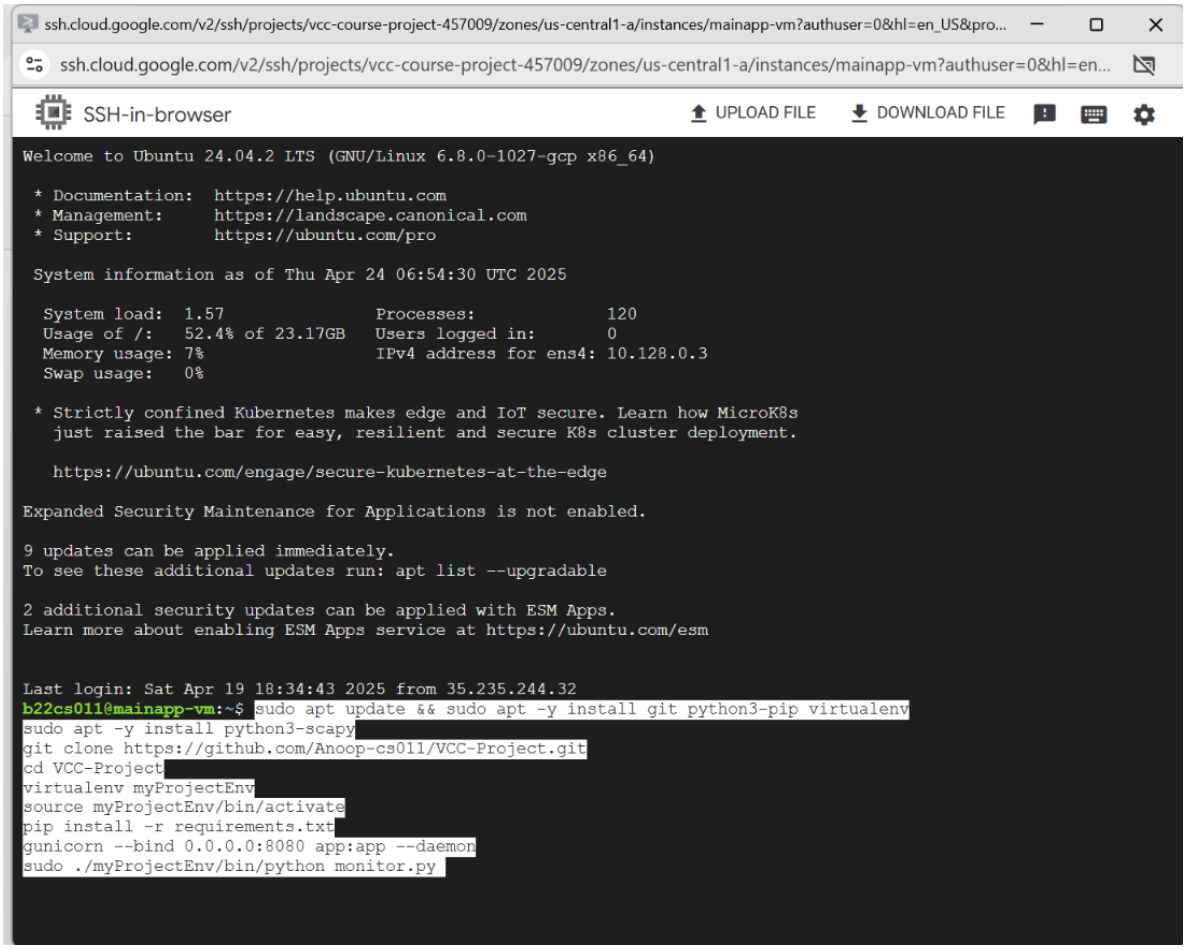
Creating the app VM with the above configurations

Running the startup script

The following shell script was executed to install the required libraries and deploy the app. This script can be found in the project’s git repository.



Start the application VM and connect to the SSH.

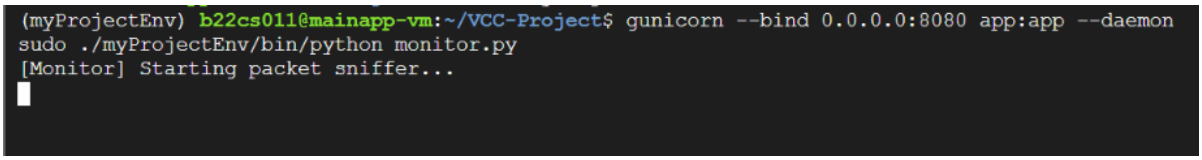


Copy and execute the shell script to deploy the app.

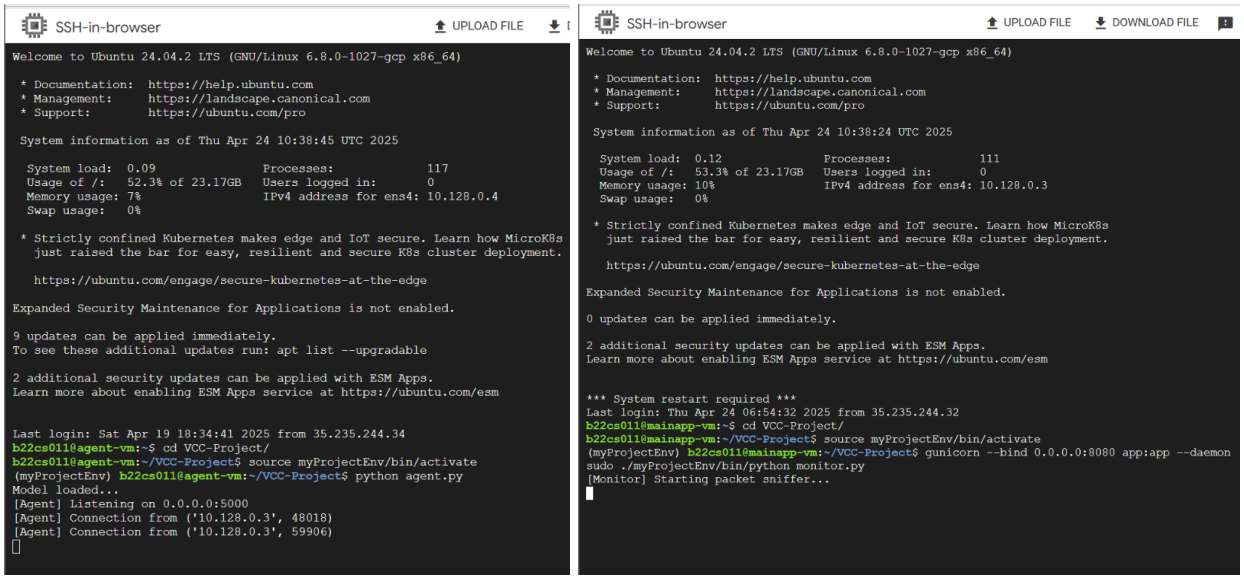
Using packet monitoring for feature extraction

The monitoring script was created in Python. It uses the 'scapy' library for packet "sniffing". Once a packet is sniffed, features are calculated and extracted from it. The features are stored until a 'batch' of packets is sniffed. Then, the 'batch' is sent to the agent VM over the network using socket programming. The prediction returned from the model is stored in a local file, which the app can read to display recent alerts.

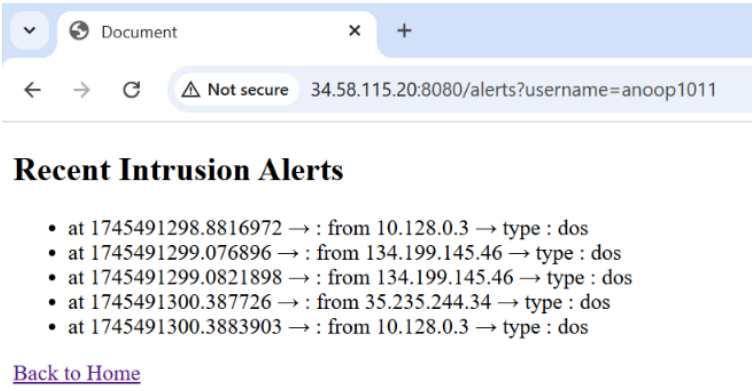
The monitoring script also begins execution once the "app" shell script is executed.



The monitoring script is also executed once the "app" shell script is executed.



The monitoring script is sniffing packets and sending batch requests to the agent VM

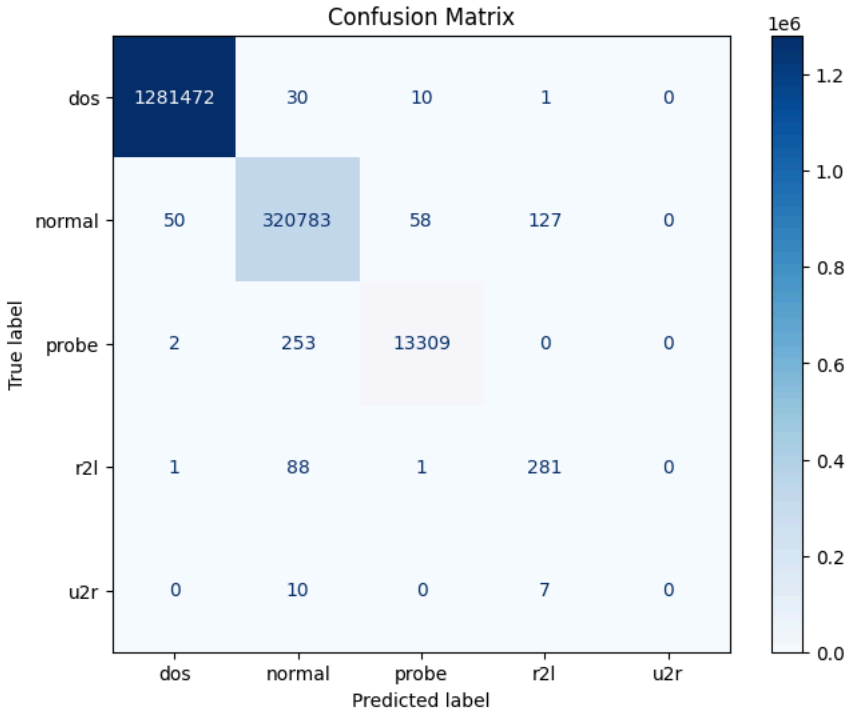


The recent intrusions can be seen from the application website after registering and logging in.

Results

DL model

As discussed earlier, the trained model was able to achieve a remarkable train accuracy of 0.9994 & validation accuracy of 0.9996 on the 5-class classification task, which is similar, if not better than the models studied from the referenced literature.



Cloud Web application deployment

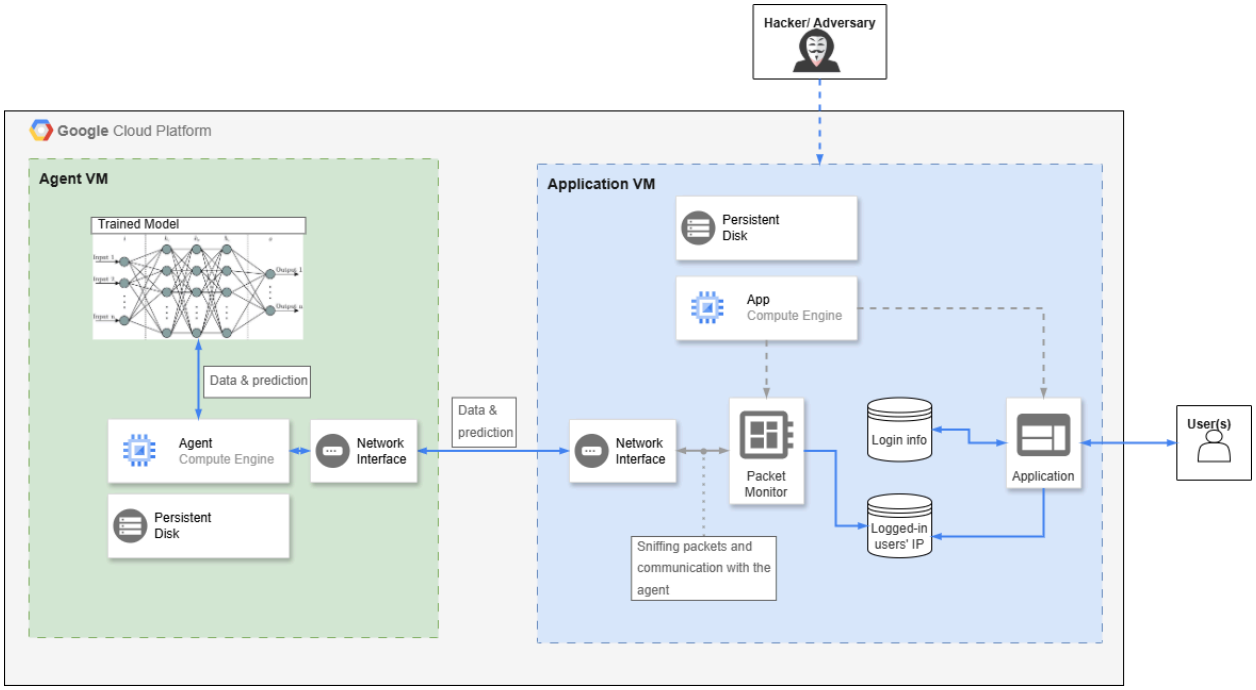
Given the high accuracy of the DL model, we were surprised to see that the model was not able to perform well on the live network data that was collected. The primary concern was that it was predicting normal network traffic as an intrusion, particularly as a DOS attack, which raised questions about the model's generalisation capability outside the dataset. This discrepancy could possibly be due to the skewed nature of the training dataset, which has an overrepresentation of the DOS samples compared to other classes. Thus, the model may have developed a bias towards DOS and begin misclassifying ambiguous or previously

unseen packets. The model probably overfits the benchmark data distribution and fails to adapt to the natural variability and noise present in real-world network traffic.

Comparison and Analysis

Such a good DL model not being able to perform well in live deployment is probably attributed to the way these models make predictions. They take one packet at a time as input and make their prediction. However, in real-world scenarios, the intrusive packets are meant to be similar to ordinary traffic. For example, in a Smurf DOS attack, ordinary ICMP echo packets are flooded into the network with a spoofed IP address of the target machine. Here, individual ICMP packets appear to be normal, but if we simply observe the abnormally large ICMP traffic, we can easily deduce that the system is under attack. Such a mechanism of detecting correlation across packets to predict intrusion can be trained using models like transformers, which utilise a cross-attention mechanism to help learn the relation between data points across the input data. These models can be trained to find patterns from a sequence of packets, rather than from one packet at a time.

Architecture Diagram



Future Scope

Future work may focus on building models that not only find patterns in individual packets but also across a sequence of packets. Using transformers might be an excellent method to realise this mechanism. However, they might be too resource-intensive, even for just inference, to keep the costs low for a feasible deployment. This could also be seen as a challenge to build a “tiny” transformer with reasonable performance in a resource-constrained environment.

The test production environment can be improved with a better design of the exploitable web application. This could be done by incorporating more features into the web application to simulate the real-world environment more accurately.

Different datasets like MSCAD & NSL-KDD can be explored and used to train and test more models. These models could possibly be used simultaneously for inference by extracting features used in multiple datasets.