The integration of large model techniques with network traffic monitoring

Xiangyu Wang,

qianyue,

BUPT wangxiangyu666@bupt.edu.cn

一、Abstract

近年来，随着网络技术的快速发展和互联网的普及，网络安全问题日益凸显。网络攻击和入侵行为的频繁发生，对信息安全构成了严重威胁。传统的网络流量监控技术已经无法满足日益复杂的网络环境和多样化的攻击手段，因此，结合大模型技术的网络流量监控逐渐成为研究的热点。

大模型技术，如深度学习和神经网络，因其强大的数据处理和模式识别能力，被广泛应用于各种领域。在网络流量监控中，利用大模型技术，可以实现对海量数据的快速分析和异常检测，提高网络安全监控的效率和准确性。

本报告将重点探讨大模型技术在网络流量监控中的应用，主要包括Linux后门检测、基于双向长短期记忆（LSTM）模型的网络攻击检测、网络入侵检测和监控工具，以及利用大型语言模型（LLM）API实现的实时网络流量分析。通过这些内容，旨在为网络安全提供更为先进和有效的技术手段。

二、Detection of Linux Backdoors

Linux是一套免费使用和自由传播的类Unix操作系统，是一个基于POSIX和Unix的多用户、多任务、支持多线程和多CPU的操作系统。它能运行主要的Unix工具软件、应用程序和网络协议，支持32位和64位硬件。Linux继承了Unix以网络为核心的设计思想，是一个性能稳定的多用户网络操作系统。Linux操作系统诞生于1991年，发展至今已经可以安装在各种计算机硬件设备中，比如手机、平板电脑、路由器、视频游戏控制台、台式计算机、大型机和超级计算机。Linux的流行让它成为众多黑客攻击的目标。

本报告以ADFA-LD数据集为例介绍Linux系统的后门检测，使用特征提取方法为2-Gram和TF-IDF，使用的分类算法包括朴素贝叶斯、XGBoost和深度学习之多层感知机。

2.1 Dataset

ADFA数据集是澳大利亚国防学院对外发布的一套主机级入侵检测系统的数据集合，被广泛应用于入侵检测类产品的测试。该数据集包括ADFA-LD和ADFA-WD，分别代表Linux系统的数据集和Windows系统的数据集。以ADFA-LD为例，它完整记录了一段时间内的操作系统的系统调用。内核提供用户空间程序与内核空间进行交互的一套标准接口，这些接口让用户态程序能受限访问硬件设备，比如申请系统资源，操作设备读写，创建新进程等。用户空间发生请求，内核空间负责执行，这些接口便是用户空间和内核空间共同识别的桥梁，这里提到两个字“受限”，是由于为了保证内核稳定性，而不能让用户空间程序随意更改系统，必须是内核对外开放的且满足权限的程序才能调用相应接口。在用户空间和内核空间之间，有一个叫做“系统调用”的中间层，是连接用户态和内核态的桥梁。这样即提高了内核的安全性，也便于移植——只需实现同一套接口即可。Linux系统，用户空间通过向内核空间发出系统调用产生软中断，从而让程序陷入内核态，执行相应的操作。对于每个系统调用都会有一个对应的系统调用号，比很多操作系统要少很多。

ADFA-LD已经将各类系统调用进行了标号，可以把这个标号理解为系统调用号。ADFA-LD针对攻击类型进行了标注，各种攻击类型见表2.1.1：

攻击类型	数据量	标注类型
Training	833	normal
Validation	4373	normal
Hydra-FTP	162	attack
Hydra-SSH	148	attack
Adduser	91	attack
Java-Meterpreter	125	attack
Meterpreter	75	attack
Webshell	118	attack

表2-1 ADFA-LD攻击类型

ADFA-LD的每个文件都独立记录了一段时间内的操作系统发生的系统调用，每个系统调用都用数字编号，对应的编号举例如下：

```
#define __NR_io_setup 0
__SYSCALL(__NR_io_setup, sys_io_setup)
#define __NR_io_destroy 1
__SYSCALL(__NR_io_destroy, sys_io_destroy)
#define __NR_io_submit 2
__SYSCALL(__NR_io_submit, sys_io_submit)
#define __NR_io_cancel 3
__SYSCALL(__NR_io_cancel, sys_io_cancel)
#define __NR_io_getevents 4
__SYSCALL(__NR_io_getevents, sys_io_getevents)
```

ADFA-LD的数据文件内容示例如图2.1.2所示，文件记录了一连串数字，是已经编号过的系统调用序列。

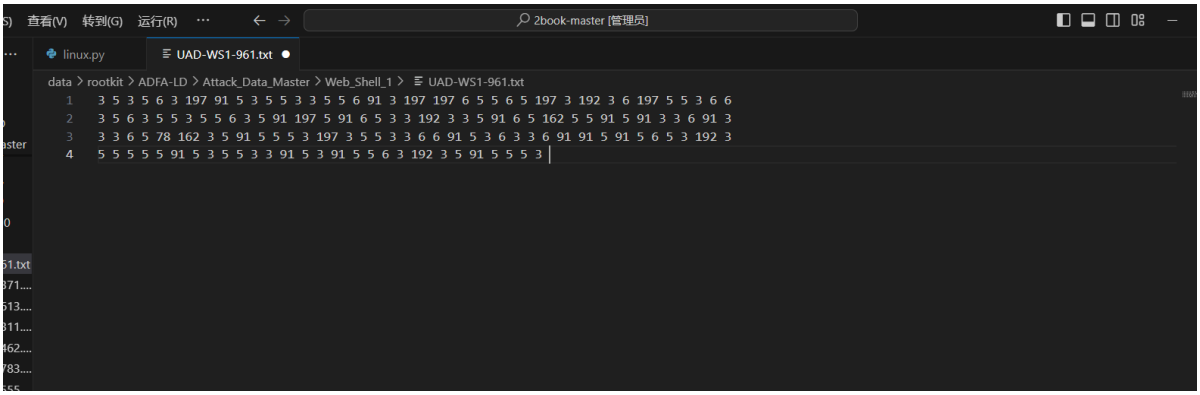


表2-2 ADFA-LD数据

2.2 Feature extraction

ADFA-LD数据集已经完成了特征提取的大部分工作，我们可以把系统调用序列当做文本处理，每个系统调用已经用数字进行了编号，把每个编号过的系统调用当做一个单词，用N-Gram进行处理后，再使用TF-IDF处理，以提高分类算法性能。加载攻击数据文件夹下面的各个数据文件，使用glob.glob函数进行枚举遍历。考虑到文件分行的问题，在将文件逐行读取完成后，逐行进行拼接，最终把一个文件加载到一个字符串变量中。使用同样的办法加载正常样本，唯一不同的是攻击样本标记为1，正常样本标记为0：

```
def load_all_files():
    x = []
    y = []
    # Load attack samples
    files = glob.glob("../data/rootkit/ADFA-LD/Attack_Data_Master/*/*")
    for file in files:
        with open(file, 'r') as f:
            lines = f.readlines()
        x.append(" ".join(lines))
        y.append(1)
    print("Load black data %d" % len(x))

    # Load normal samples
    files = glob.glob("../data/rootkit/ADFA-LD/Training_Data_Master/*")
    for file in files:
        with open(file, 'r') as f:
            lines = f.readlines()
        x.append(" ".join(lines))
        y.append(0)
    print("Load full data %d" % len(x))

    return x, y
```

加载完原始数据后，使用CountVectorizer进行N-Gram处理，这里N取3，即使用3-Gram，设置ngram_range=（3，3）即可。由于系统调用已经使用数字表示，所以单词切分使用的正则表达式可以简化为token_pattern=r'\b\d+\b'：

```
    x, y = load_all_files()
    vectorizer = CountVectorizer(
                                ngram_range=(3, 3),
                                token_pattern=r'\b\d+\b',
                                decode_error='ignore',
                                strip_accents='ascii',
                                max_features=max_features,
                                stop_words='english',
                                max_df=1.0,
                                min_df=1)
    print(vectorizer)
    x = vectorizer.fit_transform(x)
```

N-Gram处理后进行TF-IDF处理，提升分类算法的性能：

```
    transformer = TfidfTransformer(smooth_idf=False)
    x = transformer.fit_transform(x)
    x = x.toarray()
```

最后，随机划分训练数据集和测试数据集，其中测试数据集比例为40%，过高或者过低都会影响对分类器性能的判断：

```
    x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.4,
random_state=42)
    return x_train, x_test, y_train, y_test
```

2.3 Model training and validation

分别使用支持朴素贝叶斯算法、XGBoost算法、多层感知机，特征提取使用N-Gram和TF-IDF模型，完整的处理流程如图2.3.1所示。

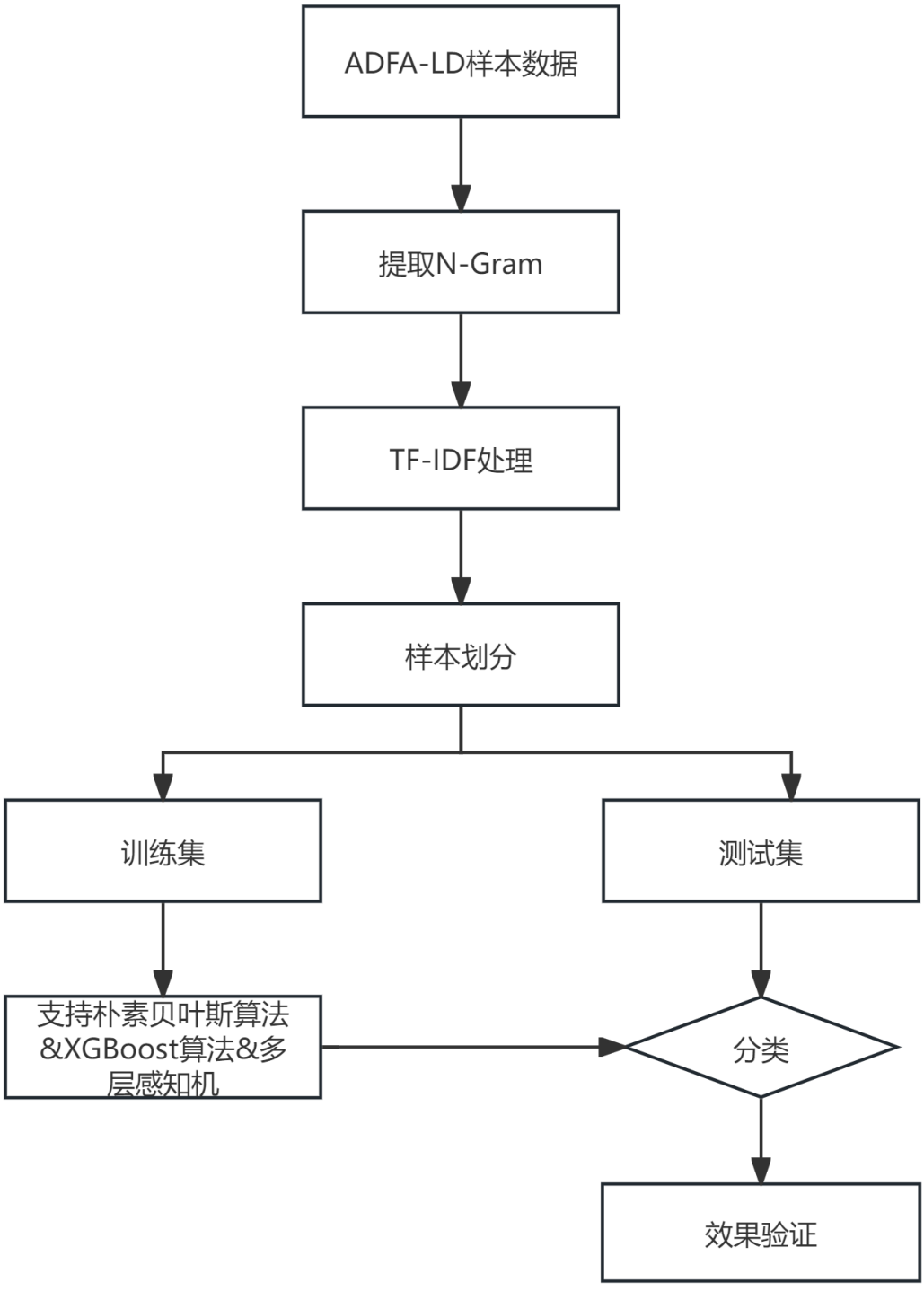


表2-3 处理流程

- 朴素贝叶斯算法代码及结果如下：

```
def do_nb(x_train, x_test, y_train, y_test):
    gnb = GaussianNB()
    gnb.fit(x_train, y_train)
    y_pred = gnb.predict(x_test)
    print(classification_report(y_test, y_pred))
    print(metrics.confusion_matrix(y_test, y_pred))
```

```
CountVectorizer(decode_error='ignore', max_features=1000, ngram_range=(3, 3),
                stop_words='english', strip_accents='ascii',
                token_pattern='\\b\\d+\\b')
              precision    recall  f1-score   support

           0       0.87      0.95      0.91       323
           1       0.95      0.84      0.89       309

    accuracy                           0.90       632
   macro avg       0.91      0.90      0.90       632
weighted avg       0.90      0.90      0.90       632

[[308  15]
 [ 48 261]]
```

表2-4 朴素贝叶斯算法

- XGBoost算法代码及结果如下：

```
def do_xgboost(x_train, x_test, y_train, y_test):
    xgb_model = xgb.XGBClassifier().fit(x_train, y_train)
    y_pred = xgb_model.predict(x_test)
    print(classification_report(y_test, y_pred))
    print(metrics.confusion_matrix(y_test, y_pred))
```

```
2-Gram & tf-idf and xgboost
              precision    recall  f1-score   support

           0       0.92      0.96      0.94       323
           1       0.95      0.92      0.93       309

    accuracy                           0.94       632
   macro avg       0.94      0.94      0.94       632
weighted avg       0.94      0.94      0.94       632

[[309  14]
 [ 26 283]]
```

表2-5 XGBoost算法

- 多层感知机代码及结果如下：

```
def do_mlp(x_train, x_test, y_train, y_test):
    # Building deep neural network
    clf = MLPClassifier(solver='lbfgs',
                        alpha=1e-5,
                        hidden_layer_sizes=(5, 2),
                        random_state=1)
    clf.fit(x_train, y_train)
    y_pred = clf.predict(x_test)
    print(classification_report(y_test, y_pred))
    print(metrics.confusion_matrix(y_test, y_pred))
```

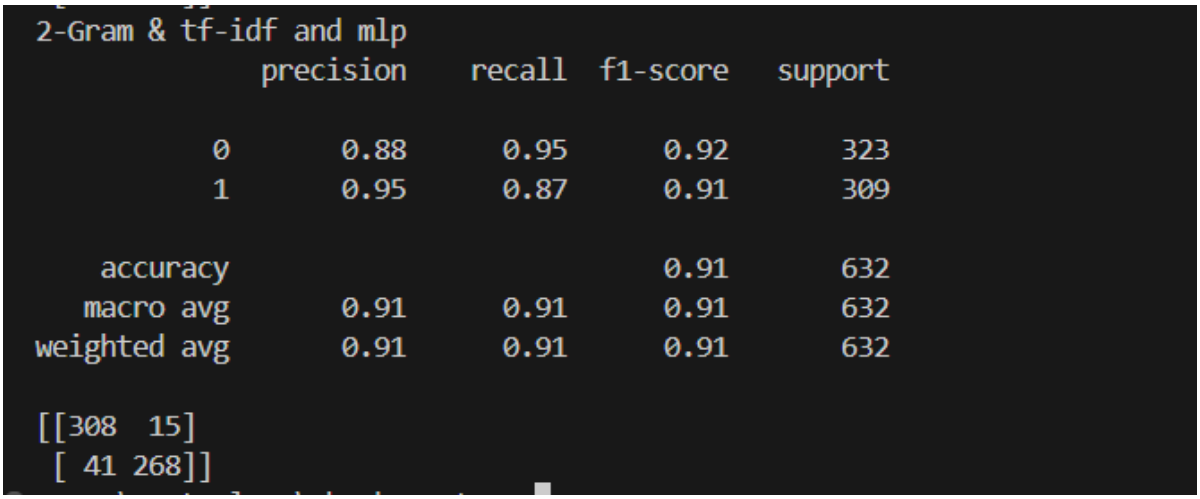


表2-6 多层感知机

三、Network Attack Detection Based on Bidirectional LSTM Models

3.1 Introduction

3.1.1 双向LSTM模型

双向LSTM（Bidirectional LSTM, BiLSTM）是在标准LSTM基础上的改进，旨在更全面地捕捉序列数据中的特征。传统的LSTM仅在时间序列的一个方向上处理数据（通常是从过去到未来），而双向LSTM在时间序列的两个方向上同时处理数据，即前向和后向。具体来说：

前向LSTM：从序列的开头到结尾依次处理数据。这个方向的LSTM可以捕捉到从过去到当前时刻的所有依赖关系。

后向LSTM：从序列的结尾到开头依次处理数据。这个方向的LSTM可以捕捉到从未来到当前时刻的所有依赖关系。

3.1.2 双向LSTM的结构

双向LSTM的结构由两个相反方向的LSTM层组成，这两个LSTM层的输出通常会在某一层进行合并或连接，形成最终的输出。具体步骤如下：

前向LSTM处理：这个LSTM从序列的第一个时间步开始，逐步处理到最后一个时间步。它输出的隐状态包含了从过去到当前时间步的所有重要信息。

后向LSTM处理：这个LSTM从序列的最后一个时间步开始，逐步处理到第一个时间步。它输出的隐状态包含了从未来到当前时间步的所有重要信息。

连接层：前向LSTM和后向LSTM的输出在时间维度上合并，通常是将两个输出向量进行拼接或求和，从而形成一个新的输出向量。这个新的输出向量同时包含了从过去到当前和从未来到当前的上下文信息。

3.1.3 双向LSTM在网络攻击检测中的具体应用步骤

- 数据预处理：

收集网络流量数据，包括正常流量和攻击流量。

提取网络流量的特征，如包长度、协议类型、IP头长度、TCP端口号等。

将数据标准化，使特征值具有相同的尺度。

- 数据格式转换：

将网络流量数据转换为适合LSTM输入的三维格式，通常包括样本数量、时间步长和特征数量。

- 模型创建：

创建一个双向LSTM模型，包含前向和后向两个LSTM层，分别处理序列数据的两个方向。

在LSTM层之后添加全连接层，用于输出分类结果。

- 模型训练：

使用训练数据集训练双向LSTM模型，通过最小化损失函数（如二元交叉熵）来优化模型参数。在训练过程中监控模型的准确率和损失，防止过拟合。

- 模型评估：

使用测试数据集评估模型的性能，计算准确率、召回率、F1分数等指标。

绘制混淆矩阵，展示模型在检测正常流量和攻击流量方面的表现。

- 模型应用：

将训练好的双向LSTM模型应用于实时网络流量检测，识别潜在的攻击行为。

根据检测结果采取相应的安全措施，如阻止恶意流量、报警等。

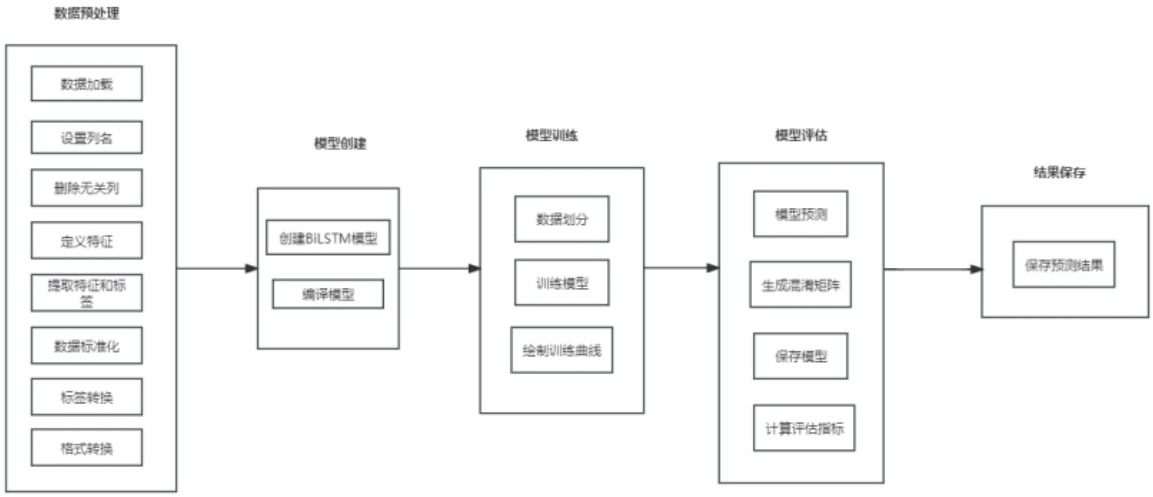


表3-1 BiLSTM攻击检测

3.2 Code Implement

- 定义常量

NUMBER_OF_SAMPLES：定义要从每个数据集中读取的样本数量。TRAIN_LEN：定义LSTM模型的输入序列长度（时间步数）。

```
NUMBER_OF_SAMPLES = 50000
TRAIN_LEN = 25
```

- 数据加载

从CSV文件中加载攻击流量和正常流量的数据集，各取50000条样本。

```
data_attack = pd.read_csv('dataset_attack.csv',nrows = NUMBER_OF_SAMPLES)
data_normal = pd.read_csv('dataset_normal.csv',nrows = NUMBER_OF_SAMPLES)
```

- 设置列名

为数据集设置明确的列名，便于后续处理和分析。

```
columns = ['frame.len', 'frame.protocols', 'ip.hdr_len', 'ip.len', 'ip.flags.rb',
'ip.flags.df', 'p.flags.mf',
           'ip.frag_offset','ip.ttl','ip.proto','ip.src','ip.dst', 'tcp.srcport',
'tcp.dstport', 'tcp.len',
            'tcp.ack', 'tcp.flags.res','tcp.flags.ns',
'tcp.flags.cwr','tcp.flags.ecn', 'tcp.flags.urg',
           'tcp.flags.ack', 'tcp.flags.push', 'tcp.flags.reset', 'tcp.flags.syn',
'tcp.flags.fin', 'tcp.window_size',
            'tcp.time_delta','class']
data_normal.columns = columns
data_attack.columns = columns
```

- 删除无关列

删除不需要的列（如IP地址和协议列），这些列可能不会对攻击检测有显著贡献

```
drop_columns =['ip.src','ip.dst','frame.protocols']
data_normal.drop(columns=drop_columns, inplace=True)
data_attack.drop(columns=drop_columns, inplace=True)
```

- 定义特征

定义用于模型训练的特征列，这些特征是网络流量中的各种统计信息。

```
features =
['frame.len','ip.hdr_len','ip.len','ip.flags.rb','ip.flags.df','p.flags.mf','ip.f
rag_offset',

'ip.ttl','ip.proto','tcp.srcport','tcp.dstport','tcp.len','tcp.ack','tcp.flags.re
s','tcp.flags.ns',

'tcp.flags.cwr','tcp.flags.ecn','tcp.flags.urg','tcp.flags.ack','tcp.flags.push',
'tcp.flags.reset',
          'tcp.flags.syn','tcp.flags.fin','tcp.window_size','tcp.time_delta']
```

- 提取特征X和标签Y

从数据集中提取特征（X）和标签（Y），并将攻击流量和正常流量的数据合并。

```
X = np.concatenate((data_normal[features].values, data_attack[features].values))
Y = np.concatenate((data_normal['class'].values, data_attack['class'].values))
```

- 标准化

对特征数据进行标准化处理，使其均值为0，标准差为1，有助于提高模型训练效果和收敛速度。

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

- 转换标签

将标签转换为二进制格式：攻击流量标签为0，正常流量标签为1。

```
Y = np.where(Y == "attack",0,1)
```

- 准备LSTM输入数据

将数据转换为适合LSTM输入的三维格式。每个输入样本包含25个时间步，每个时间步包含所有特征的数据。

```
samples =X.shape[0]
input_len=samples - TRAIN_LEN
I = np.array([X[i:i + TRAIN_LEN] for i in range(input_len)])
```

- 划分训练集和测试集

将数据集划分为训练集和测试集，测试集占20%。train_indices和test_indices记录了训练和测试集的索引，用于后续分析。

```
X_train, X_test, Y_train, Y_test, train_indices, test_indices =
train_test_split(I, Y[TRAIN_LEN:], range(len(Y[TRAIN_LEN:])), test_size = 0.2,
random_state=42)
```

- 创建模型

定义并创建一个双向LSTM模型。模型包含：一个双向LSTM层，包含64个单元，使用'tanh'激活函数和L2正则化。一个全连接层，包含128个单元，使用'relu'激活函数和L2正则化。一个输出层，包含1个单元，使用'sigmoid'激活函数和L2正则化。使用二元交叉熵作为损失函数，Adam优化器，评估指标为准确率。

```
def create_baseline():
    model = Sequential([
        Bidirectional(LSTM(64,activation='tanh',kernel_regularizer='l2')),
        Dense(128,activation='relu',kernel_regularizer='l2'),
        Dense(1,activation='sigmoid',kernel_regularizer='l2')
    ])
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=
['accuracy'])
    return model
model = create_baseline()
```

- 训练模型

使用训练集数据训练BiLSTM模型，并记录训练过程中的准确率和损失。训练过程分为5个周期（epoch），并使用20%的训练数据作为验证集。

```
history = model.fit(X_train,Y_train,epochs=5,validation_split=0.2,verbose=1)
```

- 绘制准确率和损失图

定义一个函数plot_metrics，用于绘制模型训练过程中的准确率和损失曲线图，并将其保存为图片文件。绘制并保存训练和验证集的准确率和损失曲线图。

```
def plot_metrics(history, metric, title, ylabel, save_as):
    plt.figure()
    plt.plot(history.history[metric])
    plt.plot(history.history['val_'+ metric])
    plt.title(title)
    plt.ylabel(ylabel)
    plt.xlabel('Epoch')
    plt.legend(['Train','Validation'],loc='best')
    plt.savefig(save_as)
    plt.show()
plot_metrics(history,'accuracy','BRNN Model
Accuracy','Accuracy','BRNN_Model_Accuracy.png')
plot_metrics(history,'loss','BRNN Model Loss','Loss','BRNN Model Loss.png')
```

- 模型预测

使用训练好的模型对测试集进行预测，输出预测结果并将其转换为二进制格式（0或1）。

```
predictions = model.predict(X_test,verbose=1).flatten().round()
```

- 混淆矩阵

计算并绘制混淆矩阵，展示模型在检测正常流量和攻击流量方面的表现。混淆矩阵的行和列分别表示实际标签和预测标签的频数。

```
conf_matrix = confusion_matrix(Y_test, predictions)
conf_matrix_df = pd.DataFrame(conf_matrix, index= ['Attack', 'Normal'], columns=
['Attack', 'Normal'])
sns.heatmap(conf_matrix_df, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.savefig('Confusion_Matrix_BRNN.png',dpi=400)
plt.show()
```

- 保存模型

将训练好的模型保存为文件，以便后续使用和加载。

```
model.save('brnn_model.keras')
```

- 评估模型

评估模型在测试集上的表现，计算并输出准确率和召回率。召回率是指模型在检测到的所有攻击流量中实际为攻击的比例。

```
scores = model.evaluate(X_test,Y_test, verbose=0)
recall = conf_matrix[1,1]/(conf_matrix[1,1]+ conf_matrix[1,0])
print(f"Accuracy:{scores[1]*100:.2f}%")
print(f"Recall:{recall:.2f}")
```

- 保存预测结果

将预测结果转换为DataFrame，并将0和1映射为'attack'和'normal'。提取原始数据集中测试集部分的前10列数据，并将其与预测结果合并，保存为CSV文件result.csv。

```
predictions_df = pd.DataFrame({'Predicted': predictions, 'Actual': Y_test})
predictions_df['Predicted'] = predictions_df['Predicted'].map({0: 'attack', 1:
'normal'})
predictions_df['Actual'] = predictions_df['Actual'].map({0: 'attack', 1:
'normal'})
```

- 绘制准确率和损失曲线，生成并保存混淆矩阵的热图，结果如下：

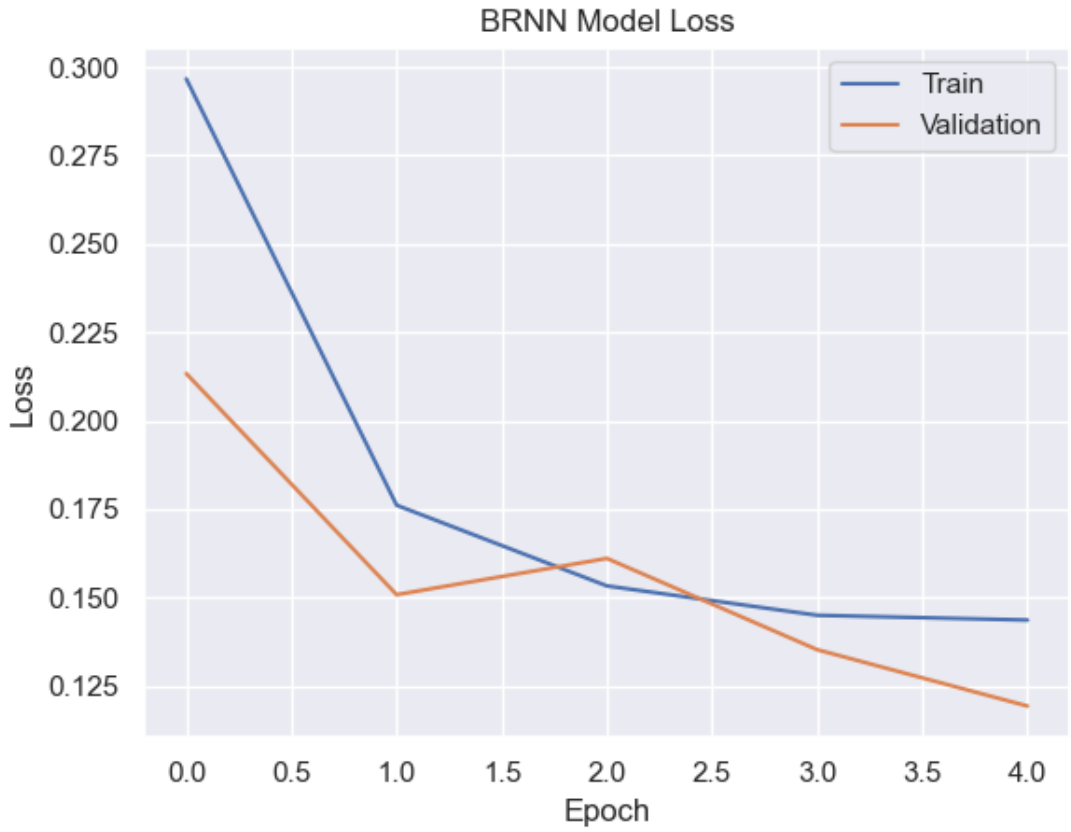


图3-2 损失曲线

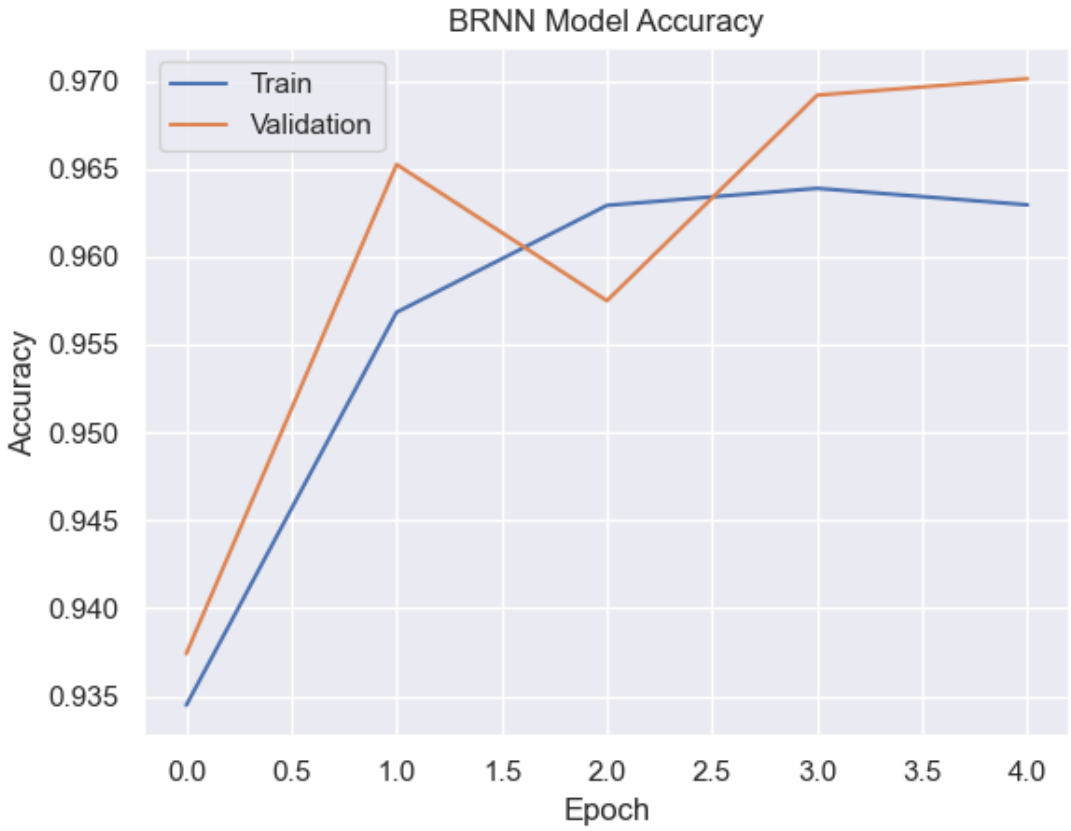


图3-3 准确率曲线

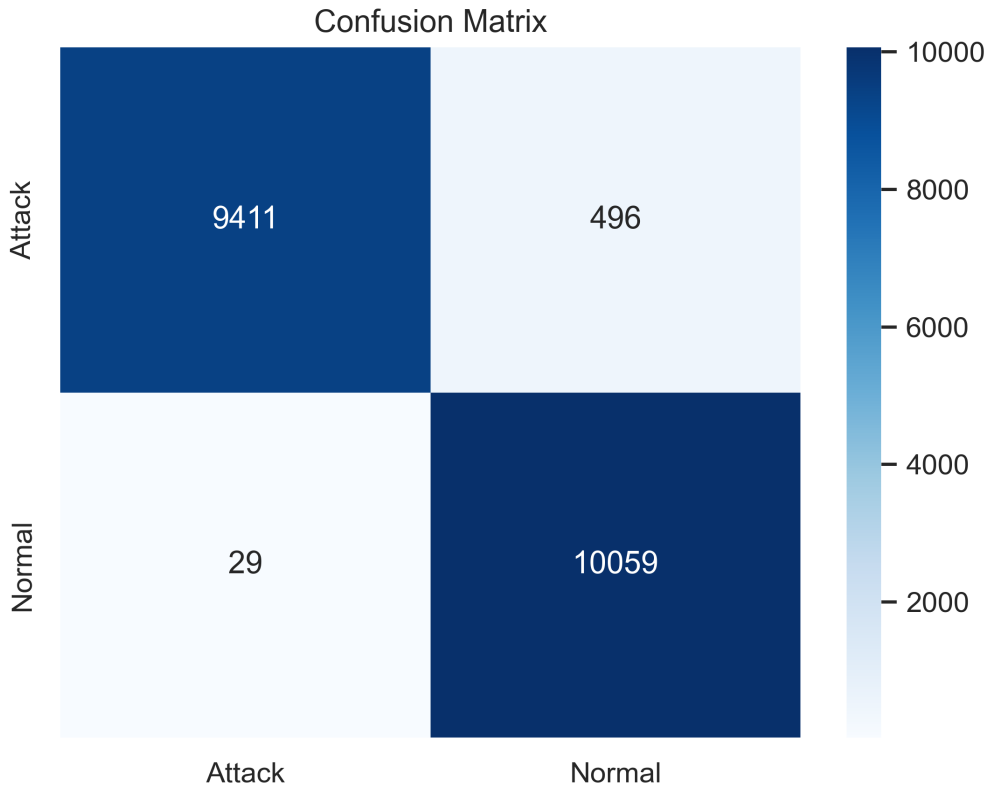


图3-4 混淆矩阵热图

四、Network-Intrusion-Detection-and-Monitoring

4.1 AIM

利用已训练好的模型，创建网络入侵检测与监控工具，在此基础上调用openai的api，实现实时的网络路由分析，综合起来，实现对机器的双向监控——一方面实时监测针对主机网络环境的入侵，另一方面实时监控主机是否访问危险违规内容。

4.2 packet_sniffer

4.2.1 常量定义和初始化

```
BUFFER_SIZE = 65536
NUMBER_OF_TENSOR = 512
TCP_PROTOCOL_NUMBER = 6
DATA_SEPARATOR = '-1'

TAB_1 = '\t - '
TAB_2 = '\t\t - '
TAB_3 = '\t\t\t - '
TAB_4 = '\t\t\t\t - '

DATA_TAB_1 = '\t'
DATA_TAB_2 = '\t\t'
DATA_TAB_3 = '\t\t\t'
DATA_TAB_4 = '\t\t\t\t'
```

- `BUFFER_SIZE`：定义了缓冲区的大小，即每次读取数据包时最多读取的字节数。
- `TCP_PROTOCOL_NUMBER`：定义了 TCP 协议的协议号（6）。
- `DATA_SEPARATOR`：用于在格式化数据时分隔不同部分的特殊字符串。
- `TAB_1` 到 `TAB_4`，`DATA_TAB_1` 到 `DATA_TAB_4`：用于打印输出时的缩进格式，方便阅读。

4.2.2 类的初始化

```
class PacketSniffer:
    def __init__(self):
        self.host = socket.gethostbyname(socket.gethostname())
        self.socket_protocol = socket.IPPROTO_IP if os.name == "nt" else
socket.IPPROTO_ICMP
        self.connection = self.__create_connection()
        self.forward_packets = 0
        self.backward_packets = 0
        self.forward_bytes = 0
        self.backward_bytes = 0
        self.start_time = time.time()
```

- `self.host`：获取本地主机的 IP 地址。
- `self.socket_protocol`：根据操作系统选择协议类型，Windows 使用 IP 协议，其它系统使用 ICMP 协议。
- `self.connection`：创建原始套接字连接。
- 初始化了一些用于流量统计的变量和开始时间戳。

4.2.3 创建连接

```
def __create_connection(self):
    connection = socket.socket(
        socket.AF_INET, socket.SOCK_RAW, self.socket_protocol)
    connection.bind((self.host, 0))
    connection.setsockopt(socket.IPPROTO_IP, socket.IP_HDRINCL, 1)
    if os.name == "nt":
        connection.ioctl(socket.SIO_RCVALL, socket.RCVALL_ON)
    return connection
```

- 创建一个原始套接字，可以捕获网络上的所有流量。
- 绑定到本地主机的 IP 地址。
- 设置套接字选项 `IP_HDRINCL`，以便在发送数据时包含 IP 头。
- 对于 Windows 系统，启用套接字的 RCVALL 模式，捕获所有进出流量。

4.2.4 嗅探数据包

```
def sniffer(self):
    raw_data, address = self.connection.recvfrom(BUFFER_SIZE)
    ip_header, ip_data = self.__ip_data_extractor(raw_data)

    if ip_header['protocol'] == TCP_PROTOCOL_NUMBER:
        tcp_header, tcp_data = self.__tcp_data_extractor(ip_data)
```

```
        flow_info, self.forward_packets, self.backward_packets,
self.forward_bytes, self.backward_bytes = self.__calculate_flow_rates(
            ip_header,
            self.host, self.start_time,
            self.forward_packets,
            self.backward_packets,
            self.forward_bytes,
            self.backward_bytes
        )

        formatted_data = self.__format_data(
            flow_info,
            tcp_header,
            ip_header,
            tcp_data
        )

        formatted_data = ' '.join(map(str, formatted_data))

        self.print_ip_data(ip_header)
        self.print_tcp_data(tcp_header, tcp_data)
        print('\n\n\n')

        return (
            formatted_data,
            flow_info,
            tcp_header['source_port'],
            tcp_header['destination_port'],
            self.__format_ipv4(ip_header['source']),
            self.__format_ipv4(ip_header['destination'])
        )
```

- `recvfrom(BUFFER_SIZE)`：从网络中接收一个数据包。
- `__ip_data_extractor(raw_data)`：提取 IP 头信息和 IP 数据。
- 检查协议是否为 TCP（6），如果是则继续处理 TCP 数据。
- `__tcp_data_extractor(ip_data)`：提取 TCP 头信息和 TCP 数据。
- `__calculate_flow_rates(...)`：计算流量统计信息，包括正向和反向的数据包和字节数，以及相应的速率。
- `__format_data(...)`：格式化提取的数据以便进一步处理或输出。
- 打印 IP 和 TCP 数据，返回格式化数据和一些关键信息。

4.2.5 提取 IP 数据

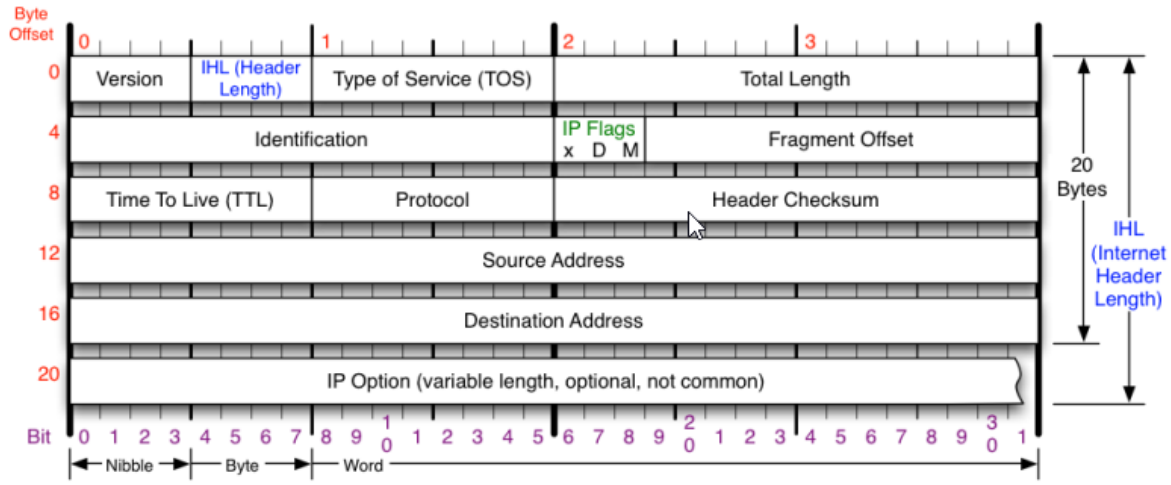


图4-1 IP数据格式

```
def __ip_data_extractor(self, data: bytes) -> Tuple[Dict[str, int], bytes]:
    version_header_len, tos, total_len, ttl, protocol, src, dst = struct.unpack(
        "! B B H 4x B B 2x 4s 4s", data[:20])
    version = version_header_len >> 4
    header_len = (version_header_len & 15) * 4
    return {
        'version': version,
        'header_length': header_len,
        'tos': tos,
        'total_length': total_len,
        'ttl': ttl,
        'protocol': protocol,
        'source': src,
        'destination': dst,
    }, data[header_len:]
```

- 使用 `struct.unpack` 从数据包中提取 IP 头字段。
- 解析 IP 头字段，如版本、头长度、服务类型（TOS）、总长度、TTL、协议、源地址和目标地址。
- 返回解析后的 IP 头信息和剩余的数据。

4.2.6 提取 TCP 数据

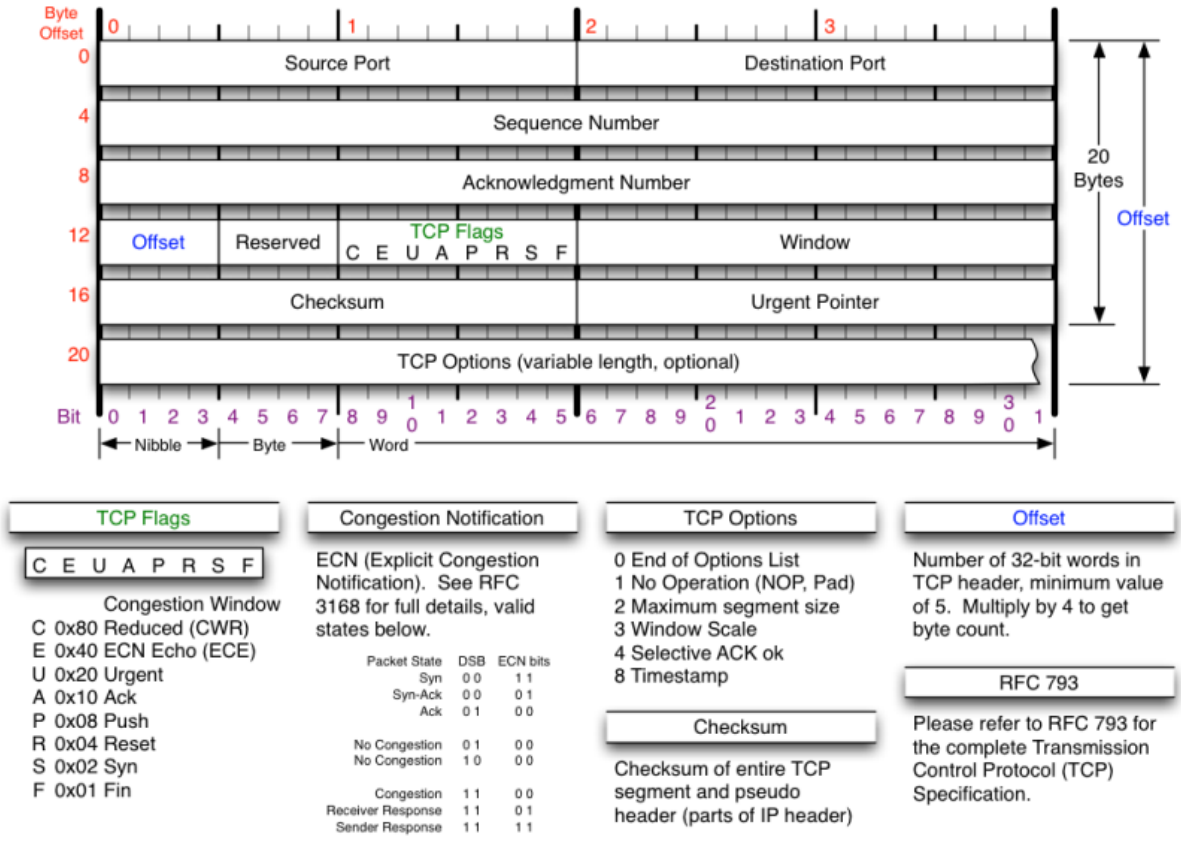


图4-2 TCP数据格式

TCP报文格式图展示了TCP头部的各个字段，包括：

1. **源端口（Source Port）和目的端口（Destination Port）**：用于标识通信的双方。
2. **序列号（Sequence Number）**：用于数据重组和丢包检测。
3. **确认号（Acknowledgment Number）**：用于确认接收到的数据。
4. **偏移（Offset）**：表示TCP头的长度。
5. **保留字段（Reserved）**：未使用，保留为未来使用。
6. **标志位（TCP Flags）**：控制数据流和连接状态的标志，如SYN、ACK、FIN等。
7. **窗口大小（Window Size）**：用于流量控制，表示接收方的缓冲区大小。
8. **校验和（Checksum）**：用于错误检测。
9. **紧急指针（Urgent Pointer）**：指出紧急数据的位置。
10. **选项（Options）**：可选字段，用于扩展TCP功能。

```
def __tcp_data_extractor(self, data: bytes) -> Tuple[Dict[str, int], bytes]:
    src_port, dst_port, offset_reserve_flag = struct.unpack(
        "! H H 8x H", data[:14])
    offset = (offset_reserve_flag >> 12) * 4
    flag_urg = (offset_reserve_flag & 32) >> 5
    flag_ack = (offset_reserve_flag & 16) >> 4
    flag_psh = (offset_reserve_flag & 8) >> 3
    flag_rst = (offset_reserve_flag & 4) >> 2
    flag_syn = (offset_reserve_flag & 2) >> 1
    flag_fin = offset_reserve_flag & 1
```

```
    return {
        'source_port': src_port,
        'destination_port': dst_port,
        'offset': offset,
        'flag_urg': flag_urg,
        'flag_ack': flag_ack,
        'flag_psh': flag_psh,
        'flag_rst': flag_rst,
        'flag_syn': flag_syn,
        'flag_fin': flag_fin,
    }, data[offset:]
```

- 使用 `struct.unpack` 从 IP 数据中提取 TCP 头字段。
- 解析 TCP 头字段，如源端口、目标端口、数据偏移量和标志位（URG、ACK、PSH、RST、SYN、FIN）。
- 返回解析后的 TCP 头信息和剩余的 TCP 数据。

4.2.7 数据格式化和流量计算

```
def __format_data(self,
                  flow_info,
                  tcp_header,
                  ip_header,
                  tcp_data):
    formatted_data = [
        int(flow_info['fpps']),
        int(flow_info['bpps']),
        int(flow_info['btps']),
        DATA_SEPARATOR,
        str(tcp_header['source_port']),
        str(tcp_header['destination_port']),
        str(ip_header['total_length']),
        str(len(tcp_data)),
        str(ip_header['ttl']),
        str(ip_header['tos']),
        str(tcp_header['offset']),
        str(tcp_header['flag_urg'] |
            tcp_header['flag_ack'] << 1 |
            tcp_header['flag_psh'] << 2 |
            tcp_header['flag_rst'] << 3 |
            tcp_header['flag_syn'] << 4 |
            tcp_header['flag_fin'] << 5
            )
    ]

    raw_bytes = [str(byte) for byte in tcp_data]

    formatted_data.append(DATA_SEPARATOR)
    formatted_data.extend(raw_bytes)

    return formatted_data

def __calculate_flow_rates(self,
                           ip_header: Dict[str, int],
```

```
                            host: str, start_time: float,
                            forward_packets: int,
                            backward_packets: int,
                            forward_bytes: int,
                            backward_bytes: int) -> Tuple[Dict[str, float], int,
int, int, int]:
    if self.__format_ipv4(ip_header['source']) == host:
        forward_packets += 1
        forward_bytes += ip_header["total_length"]
    elif self.__format_ipv4(ip_header['destination']) == host:
        backward_packets += 1
        backward_bytes += ip_header["total_length"]

    current_time = time.time()
    elapsed_time = current_time - start_time  # in seconds

    backward_forward_bytes = backward_bytes + forward_bytes

    forward_packets_per_second = forward_packets / elapsed_time
    backward_packets_per_second = backward_packets / elapsed_time
    bytes_transferred_per_second = backward_forward_bytes / elapsed_time

    return {
        'fpps': forward_packets_per_second,
        'bpps': backward_packets_per_second,
        'btps': bytes_transferred_per_second
    }, forward_packets, backward_packets, forward_bytes, backward_bytes
```

- `__format_data`：格式化提取的 IP 和 TCP 头信息以及 TCP 数据，返回一个列表。
- `__calculate_flow_rates`：计算数据包的流速，包括每秒正向和反向的数据包数、传输的字节数。

4.3 network_detection_model

- 使用已训练好的AI模型，模型来源：rdpahalavan/bert-network-packet-flow-header-payload · Hugging Face。此模型经过微调，可将网络数据包分类为以下类别之一。

```
class NetworkDetectionModel:
    CLASS_LABELS = [
        'Analysis',
        'Backdoor',
        'Bot',
        'DDoS',
        'DoS',
        'DoS GoldenEye',
        'DoS Hulk',
        'DoS SlowHTTPTest',
        'DoS Slowloris',
        'Exploits',
        'FTP Patator',
        'Fuzzers',
        'Generic',
        'Heartbleed',
        'Infiltration',
```

```
        'Normal',
        'Port Scan',
        'Reconnaissance',
        'SSH Patator',
        'Shellcode',
        'Web Attack - Brute Force',
        'Web Attack - SQL Injection',
        'Web Attack - XSS',
        'Worms'
    ]
```

- 训练所使用的数据集及数据集格式如下：

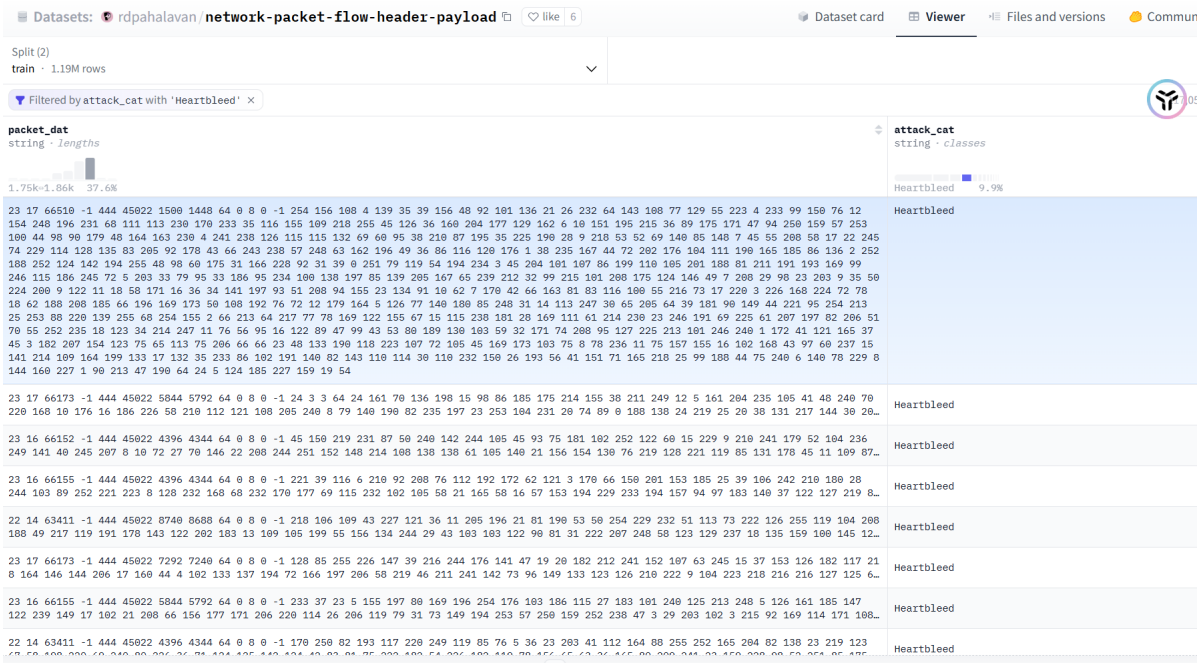


图4-3 数据集

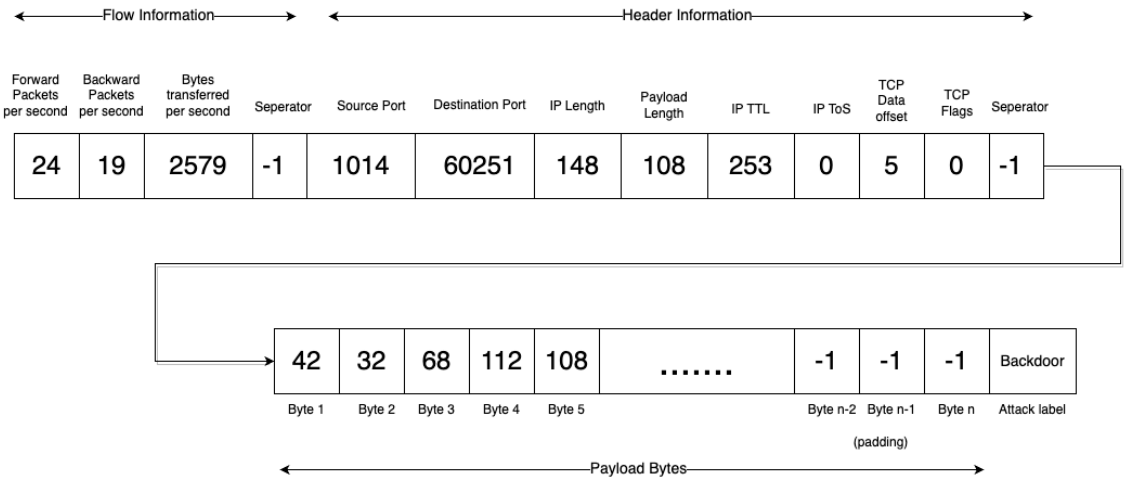


图4-4 数据集格式

- **预测方法** `predict`：
  - 接受一个字符串序列作为输入。
  - 对输入序列进行分词，确保分词后的长度不超过最大限制（512减去2个特殊标记的长度）。
  - 将分词后的序列转换回字符串，再进行一次分词并转换为张量。
  - 使用模型进行分类，得到输出结果。

- 取出模型输出中概率最高的类别索引，并转换为类别标签。

```
    def predict(self, sequence: str) -> str:
        # Tokenize the sequence
        tokens = self.tokenizer.tokenize(sequence)

        # Ensure the number of tokens doesn't exceed the maximum
        MAX_TOKENS = 512 - 2  # subtract 2 for [CLS] and [SEP]
        tokens = tokens[:MAX_TOKENS]

        # Convert the tokens back to a string
        sequence = self.tokenizer.convert_tokens_to_string(tokens)

        # Tokenize the sequence again, this time with return_tensors="pt"
        inputs = self.tokenizer(sequence, return_tensors="pt")

        # Perform the classification
        outputs = self.model(**inputs)

        # The model returns first element with highest probability
        logits = outputs[0]

        # To get the predicted class
        predicted_class_index = logits.argmax(dim=-1).item()
        predicted_class_label = self.CLASS_LABELS[predicted_class_index]

        return predicted_class_label
```

- 调用网络流量监控和分析脚本，使用自定义的 `PacketSniffer` 和 `NetworkDetectionModel` 类来捕获网络数据包并进行分析。根据分析结果，脚本会将源IP地址分类为 `ALLOW`、`ALERT` 或 `BLOCK`。如果一个IP在过去一小时内被阻止了三次或更多次，将会采取进一步的阻止措施。

```
from packet_sniffer import PacketSniffer
from network_detection_model import NetworkDetectionModel
import csv
from datetime import datetime
import socket
import os
import pandas as pd
from util import block_ip, alert_ip


SCRIPT_DIR = os.path.dirname(os.path.realpath(__file__))


ALLOW = ['Normal']

ALERT = ['Analysis', 'Port Scan', 'Reconnaissance']

BLOCK = [
    'Backdoor',
    'Bot',
    'DDoS',
    'DoS',
```

```
    'DoS GoldenEye',
    'DoS Hulk',
    'DoS SlowHTTPTest',
    'DoS Slowloris',
    'Exploits',
    'FTP Patator',
    'Fuzzers',
    'Generic',
    'Heartbleed',
    'Infiltration',
    'SSH Patator',
    'Shellcode',
    'Web Attack - Brute Force',
    'Web Attack - SQL Injection',
    'Web Attack - XSS',
    'Worms'
]


def get_domain_name(ip):
    try:
        return socket.gethostbyaddr(ip)[0]
    except socket.herror:
        return ''


def count_blocks_in_last_hour(ip):
    log_file_path = os.path.join(SCRIPT_DIR, 'log_data/log.csv')
    log_df = pd.read_csv(log_file_path)

    # Filter the rows where the source IP is the given IP and the status is
'BLOCK'
    ip_blocks = log_df[(log_df['source_ip'] == ip) &
                       (log_df['status'] == 'BLOCK')]

    # Convert the 'date' column to datetime
    ip_blocks['date'] = pd.to_datetime(ip_blocks['date'])

    # Filter the rows in the last hour
    ip_blocks_last_hour = ip_blocks[ip_blocks['date']
                                    > datetime.now() - pd.Timedelta(hours=1)]

    # Return the count of rows
    return len(ip_blocks_last_hour)


def main():
    packet_sniffer = PacketSniffer()
    network_detection = NetworkDetectionModel()

    log_file_path = os.path.join(SCRIPT_DIR, 'log_data/log.csv')
    log_file = open(log_file_path, 'a', newline='')
    log_writer = csv.writer(log_file)

    unique_addr_file_path = os.path.join(SCRIPT_DIR, 'log_data/unique.csv')
    unique_addr_file = open(unique_addr_file_path, 'a', newline='')
```

```
    unique_addr_writer = csv.writer(unique_addr_file)

    unique_addr = pd.read_csv(unique_addr_file_path)
    blocked_ips = unique_addr.loc[unique_addr['status']== 'blocked',
'source_ip'].tolist()

    # Create a set to store unique addresses
    unique_addresses = set(unique_addr['source_ip'].tolist())

    while True:
        data = packet_sniffer.sniffer()
        if data == None:
            continue

        formatted_data, flow_info, source_port, destination_port, source_ip,
destination_ip = data
        if source_ip == packet_sniffer.host or source_ip in blocked_ips:
            continue

        prediction = network_detection.predict(formatted_data)
        date = datetime.now().strftime('%Y-%m-%d %H:%M:%S')
        # domain = get_domain_name(source_ip)
        domain = ''
        if prediction in ALLOW:
            log_writer.writerow(
                [
                    date,
                    domain,
                    source_ip,
                    source_port,
                    destination_ip,
                    destination_port,
                    'ALLOW',
                    prediction
                ]
            )
            log_file.flush()

            if source_ip not in unique_addresses:
                unique_addr_writer.writerow([date,domain,source_ip,'ALLOW'])
                unique_addresses.add(source_ip)
                unique_addr_file.flush()

        elif prediction in ALERT:
            # alert(ip)
            log_writer.writerow(
                [
                    date,
                    domain,
                    source_ip,
                    source_port,
                    destination_ip,
                    destination_port,
                    'ALERT',
                    prediction
                ]
```

```
            )
            log_file.flush()

            if source_ip not in unique_addresses:
                unique_addr_writer.writerow([date,domain,source_ip,'ALERT'])
                unique_addresses.add(source_ip)
                unique_addr_file.flush()

        elif prediction in BLOCK:
            log_writer.writerow(
                [
                    date,
                    domain,
                    source_ip,
                    source_port,
                    destination_ip,
                    destination_port,
                    'BLOCK',
                    prediction
                ]
            )
            log_file.flush()

            if count_blocks_in_last_hour(source_ip) >= 3:
                # block_ip(source_ip)
                unique_addresses.add(source_ip)
                if source_ip not in unique_addresses:
                    unique_addr_writer.writerow([date, domain, source_ip,
'BLOCK'])
                    unique_addr_file.flush()
                else:
                    unique_addr_df = pd.read_csv(unique_addr_file_path)
                    unique_addr_df.loc[unique_addr_df['source_ip'] == source_ip,
'status'] = 'BLOCK'
                    unique_addr_df.to_csv(unique_addr_file_path, index=False)


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

4.4 RealTimeLLMNetworkAnalysis

使用Scapy库进行网络数据包捕获，并使用OpenAI的API对查询的域名进行分类。

- 初始化定义

```
openai.api_key = os.getenv("OPENAI_API_KEY")

log_file_path = 'dns_log.csv'  # Update this path as needed

domain_category_cache = {}  # Cache for domain categories to minimize API
requests
query_count_cache = {}  # Cache for counting domain queries
```

- 初始化CSV文件

```
def initialize_csv():
    """Initialize the CSV log file with headers if it doesn't already exist."""
    headers = ['Timestamp', 'Queried Domain', 'Device IP', 'Source MAC',
'Destination IP', 'Destination MAC', 'Protocol', 'Packet Size', 'DNS Response
Code', 'Category', 'Query Count']
    if not os.path.exists(log_file_path):
        with open(log_file_path, 'w', newline='') as csvfile:
            writer = csv.writer(csvfile)
            writer.writerow(headers)
    else:
        print("CSV file already exists. Continuing with the existing file.")
```

- 域名分类函数

使用OpenAI的API对域名进行分类，并将结果缓存以减少API请求次数。

```
def categorize_domain(domain_name):
    """Categorize the domain name using OpenAI's API."""
    if domain_name in domain_category_cache:
        return domain_category_cache[domain_name]

    try:
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo-instruct",  # Adjust for the latest model as
necessary
            prompt=f"What category best describes the website with the domain
name: {domain_name}?",
            temperature=0.5,
            max_tokens=10)
        category = response.choices[0].text.strip()
        domain_category_cache[domain_name] = category
        return category
    except Exception as e:
        print(f"An error occurred while categorizing domain: {e}")
        return "Error"
```

- 更新查询计数函数

  更新每个域名的查询次数，并将结果缓存。

```
def update_query_count(domain_name):
    """Update the query count for a given domain name."""
    if domain_name in query_count_cache:
        query_count_cache[domain_name] += 1
    else:
        query_count_cache[domain_name] = 1
    return query_count_cache[domain_name]
```

- 处理数据包函数

处理捕获的数据包，提取DNS查询信息，并调用分类和计数更新函数。

```
def process_packet(packet):
    """Process each packet captured, extract DNS query information, and
categorize domain."""
```

```
    if DNSQR in packet and packet.haslayer(DNS):
        queried_domain = packet[DNSQR].qname.decode('utf-8').rstrip('.')
        category = categorize_domain(queried_domain)
        query_count = update_query_count(queried_domain)

        packet_details = {
            'device_ip': packet[IP].src if IP in packet else (packet[IPv6].src if
IPv6 in packet else 'N/A'),
            'source_mac': packet[Ether].src,
            'destination_ip': packet[IP].dst if IP in packet else
(packet[IPv6].dst if IPv6 in packet else 'N/A'),
            'destination_mac': packet[Ether].dst,
            'protocol': 'TCP' if TCP in packet else ('UDP' if UDP in packet else
'N/A'),
            'packet_size': len(packet),
            'dns_response_code': packet[DNS].rcode if packet[DNS].qr == 1 and
packet[DNS].ancount > 0 else 'N/A',
            'category': category,
            'query_count': query_count
        }

        timestamp = datetime.now().strftime('%Y-%m-%d %H:%M:%S')

        log_visit(timestamp, queried_domain, **packet_details)
```

- 记录查询日志

将每个DNS查询的详细信息记录到CSV文件中。

```
def log_visit(timestamp, queried_domain, device_ip, source_mac, destination_ip,
destination_mac, protocol, packet_size, dns_response_code, category,
query_count):
    with open(log_file_path, 'a', newline='') as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow([timestamp, queried_domain, device_ip, source_mac,
destination_ip, destination_mac, protocol, packet_size, dns_response_code,
category, query_count])
```

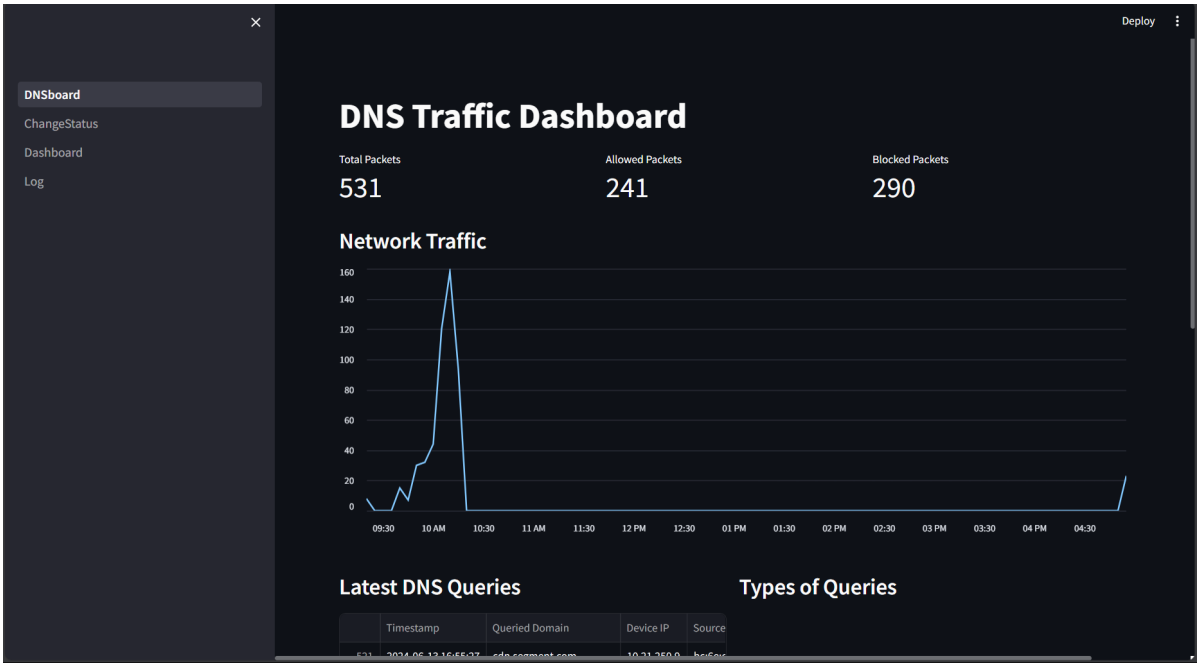
- 嗅探结果整理调用api生成个性化分析报告

```
def generate_summary_report(df):
    """生成网络活动的总结报告"""
    allowed_packets = len(df[df['DNS Response Code'] == 0])
    blocked_packets = len(df[df['DNS Response Code'] != 0])
    most_common_domain = df['Queried Domain'].value_counts().idxmax() if not
df['Queried Domain'].empty else "N/A"
    most_common_category = df['Category'].value_counts().idxmax() if not
df['Category'].empty else "N/A"

    summary = f"""
    **Activity Summary:**

    - Total Packets: {len(df)}
    - Allowed Packets: {allowed_packets}
```

```
    - Blocked Packets: {blocked_packets}
    - Most Queried Domain: {most_common_domain}
    - Most Common Category: {most_common_category}
    """

    try:
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user", "content": f"Generate a summary report based on
the following data:\n{summary}"}
            ]
        )
        return response.choices[0].message['content'].strip()
    except Exception as e:
        st.error(f"Error generating summary report: {e}")
        return summary
```

4.5 Intergrated on streamlit

前端交互分四个部分：DNSboard、Dashboard、ChangeStatus、Log



4.5.1 DNSboard

```
import streamlit as st
import pandas as pd
from datetime import datetime
import plotly.express as px
import plotly.graph_objects as go
import os
import openai

# 设置OpenAI API密钥
openai.api_key = os.getenv("OPENAI_API_KEY")

# 设定日志文件路径
```

```
log_file_path = 'log_data/dns_log.csv'  # 更新路径

st.set_page_config(page_title="DNS Dashboard", page_icon=":bar_chart:",
layout="wide")

def load_data():
    """加载CSV文件数据"""
    if os.path.exists(log_file_path):
        try:
            df = pd.read_csv(log_file_path, parse_dates=['Timestamp'], na_values=
['', 'NA', 'NaN'])
            return df
        except Exception as e:
            st.error(f"Error reading the log file: {e}")
            return pd.DataFrame()
    else:
        st.error(f"Log file {log_file_path} does not exist.")
        return pd.DataFrame()

def get_domain_analysis(domain):
    """使用OpenAI API对域名进行详细分析和解释"""
    try:
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user", "content": f"Analyze the following domain for
potential threats and unusual activity: {domain}. Provide a detailed explanation
and recommendations."}
            ]
        )
        return response.choices[0].message['content'].strip()
    except Exception as e:
        st.error(f"Error analyzing domain: {e}")
        return "Error"

def generate_summary_report(df):
    """生成网络活动的总结报告"""
    allowed_packets = len(df[df['DNS Response Code'] == 0])
    blocked_packets = len(df[df['DNS Response Code'] != 0])
    most_common_domain = df['Queried Domain'].value_counts().idxmax() if not
df['Queried Domain'].empty else "N/A"
    most_common_category = df['Category'].value_counts().idxmax() if not
df['Category'].empty else "N/A"

    summary = f"""
    **Activity Summary:**

    - Total Packets: {len(df)}
    - Allowed Packets: {allowed_packets}
    - Blocked Packets: {blocked_packets}
    - Most Queried Domain: {most_common_domain}
    - Most Common Category: {most_common_category}
    """

    try:
```

```
        response = openai.ChatCompletion.create(
            model="gpt-3.5-turbo",
            messages=[
                {"role": "system", "content": "You are a helpful assistant."},
                {"role": "user", "content": f"Generate a summary report based on
the following data:\n{summary}"}
            ]
        )
        return response.choices[0].message['content'].strip()
    except Exception as e:
        st.error(f"Error generating summary report: {e}")
        return summary

def app():
    st.title('DNS Traffic Dashboard')

    df = load_data()

    if df.empty:
        st.write("No data available.")
        return

    # 显示统计信息
    total_packets = len(df)
    allowed_packets = len(df[df['DNS Response Code'] == 0])
    blocked_packets = len(df[df['DNS Response Code'] != 0])

    total_card, allowed_card, blocked_card = st.columns(3)
    total_card.metric("Total Packets", total_packets)
    allowed_card.metric("Allowed Packets", allowed_packets)
    blocked_card.metric("Blocked Packets", blocked_packets)

    # 绘制整个数据集的网络流量折线图
    df_resampled = df.set_index('Timestamp').resample('5Min').count()

    st.subheader('Network Traffic')
    st.line_chart(df_resampled['Queried Domain'])

    # 最新DNS查询表格和攻击类型饼图
    latest_queries_table, query_type_pie = st.columns(2)

    latest_queries_table.subheader('Latest DNS Queries')
    latest_queries_table.dataframe(df.tail(10)[['Timestamp', 'Queried Domain',
'Device IP', 'Source MAC', 'Destination IP', 'Destination MAC', 'Protocol',
'Packet Size', 'DNS Response Code', 'Category', 'Query Count']])

    query_type_pie.subheader('Types of Queries')

    # 大类数据
    df['Main Category'] = df['Category'].apply(lambda x: x.split('/')[0] if
pd.notnull(x) else "Unknown")  # 生成大类
    df_main_categories = df['Main Category'].value_counts()

    fig = go.Figure()

    # 添加大类饼图
```

```
    fig.add_trace(go.Pie(labels=df_main_categories.index,
values=df_main_categories, name="Main Category"))

    # 添加小类饼图（初始不可见）
    for main_category in df_main_categories.index:
        subcategories = df[df['Main Category'] == main_category]
['Category'].value_counts()
        fig.add_trace(go.Pie(labels=subcategories.index, values=subcategories,
name=main_category, visible=False))

    # 创建按钮来切换视图
    buttons = []
    buttons.append(dict(label='Main Categories',
                        method='update',
                        args=[{'visible': [True] + [False] *
len(df_main_categories.index)},
                              {'title': 'Main Categories'}]))

    for i, main_category in enumerate(df_main_categories.index):
        visible_array = [False] * (len(df_main_categories.index) +
len(df_main_categories.index))
        visible_array[i + 1] = True
        buttons.append(dict(label=main_category,
                            method='update',
                            args=[{'visible': visible_array},
                                  {'title': main_category}]))

    fig.update_layout(updatemenus=[dict(type='dropdown', direction='down',
buttons=buttons, x=1, y=1, showactive=True)])

    query_type_pie.plotly_chart(fig)

    # 域名查询和分析功能
    st.subheader('Domain Lookup and Analysis')
    domain_lookup = st.text_input('Enter a domain to lookup and analyze', '')
    if domain_lookup:
        domain_info = df[df['Queried Domain'].str.contains(domain_lookup,
case=False, na=False)]
        if not domain_info.empty:
            st.write(f"Results for '{domain_lookup}':")
            st.dataframe(domain_info[['Timestamp', 'Queried Domain', 'Device IP',
'Source MAC', 'Destination IP', 'Destination MAC', 'Protocol', 'Packet Size',
'DNS Response Code', 'Category', 'Query Count']])
            st.write("Detailed Analysis from OpenAI:")
            domain_analysis = get_domain_analysis(domain_lookup)
            st.write(domain_analysis)
        else:
            st.write(f"No results found for '{domain_lookup}'.")

    # 生成报告
    st.subheader('Generate Summary Report')
    if st.button('Generate Summary Report'):
        summary_report = generate_summary_report(df)
        st.markdown(summary_report)

if __name__ == '__main__':
```

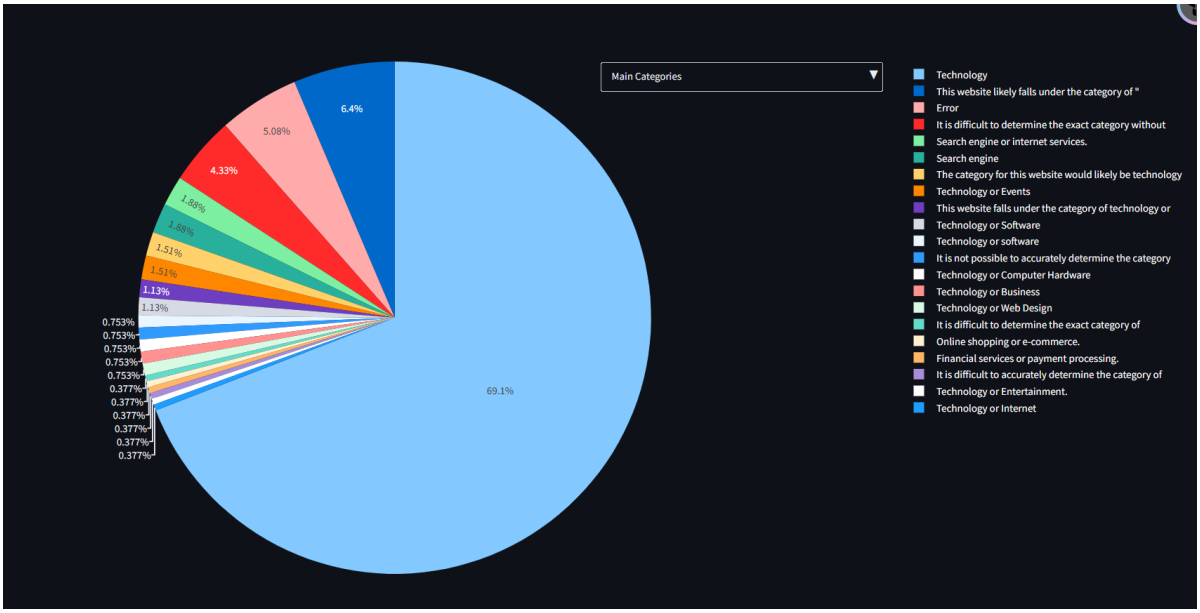
```
app()
```

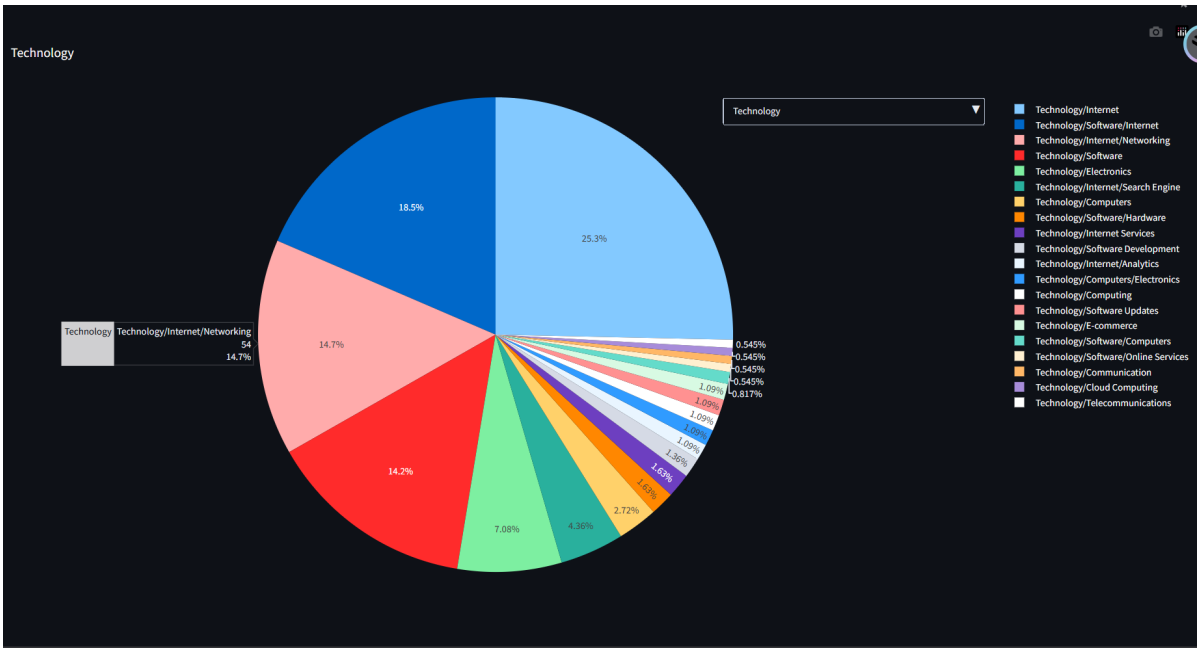
- 绘制整个数据集的网络流量折线图

- 显示统计信息

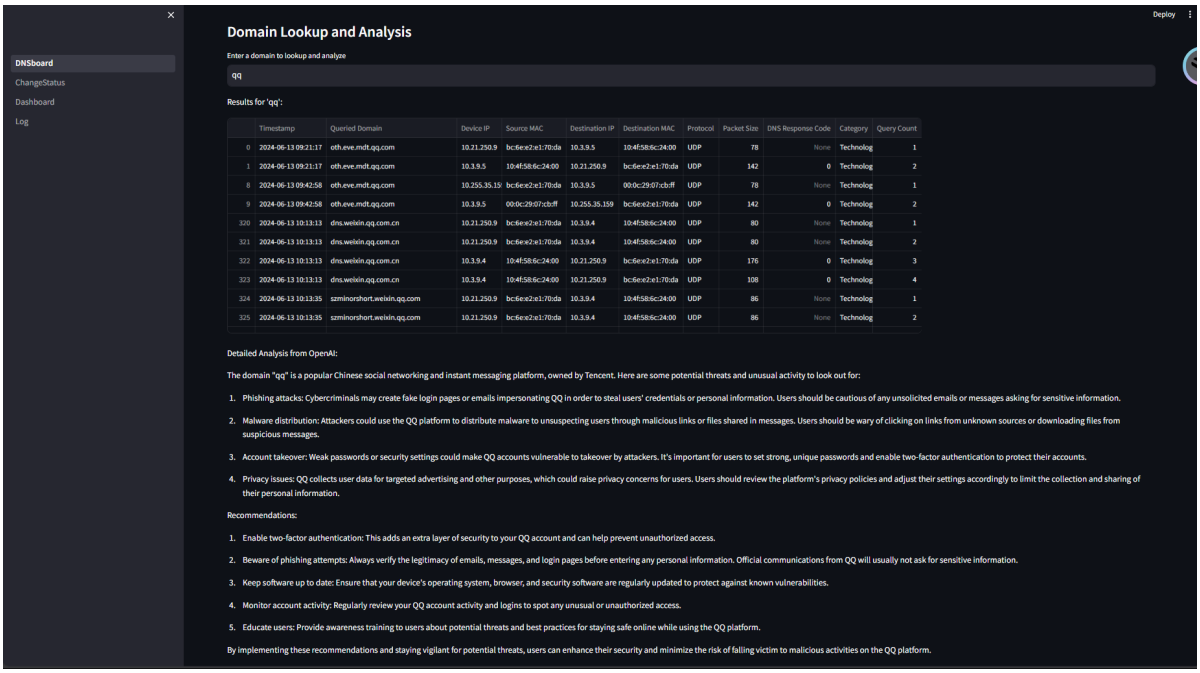
	Timestamp	Queried Domain	Device IP	Source MAC	Destination...	Destination ...	Proto...	Packet S...	DNS Response C...	Category
2	2024-06-13 09:21:17	oth.eve.mdt.qq.com	10.21.250.9	bc:6e:e2:e1:70:da	10.3.9.5	10:4f:58:6c:24:00	UDP	78	N/A	Technology/Internet
3	2024-06-13 09:21:17	oth.eve.mdt.qq.com	10.3.9.5	10:4f:58:6c:24:00	10.21.250.9	bc:6e:e2:e1:70:...	UDP	142	0	Technology/Internet
4	2024-06-13 09:21:25	get.sogou.com	10.21.250.9	bc:6e:e2:e1:70:da	10.3.9.5	10:4f:58:6c:24:00	UDP	73	N/A	Search engine or internet services.
5	2024-06-13 09:21:25	get.sogou.com	10.21.250.9	bc:6e:e2:e1:70:da	10.3.9.5	10:4f:58:6c:24:00	UDP	73	N/A	Search engine or internet services.
6	2024-06-13 09:21:25	get.sogou.com	10.3.9.5	10:4f:58:6c:24:00	10.21.250.9	bc:6e:e2:e1:70:...	UDP	199	0	Search engine or internet services.
7	2024-06-13 09:21:25	get.sogou.com	10.3.9.5	10:4f:58:6c:24:00	10.21.250.9	bc:6e:e2:e1:70:...	UDP	169	0	Search engine or internet services.
8	2024-06-13 09:21:26	365.kdocs.cn	10.21.250.9	bc:6e:e2:e1:70:da	223.5.5.5	10:4f:58:6c:24:00	UDP	72	N/A	This website likely falls under the categ
9	2024-06-13 09:21:31	365.kdocs.cn	223.5.5.5	10:4f:58:6c:24:00	10.21.250.9	bc:6e:e2:e1:70:...	UDP	104	0	This website likely falls under the categ
10	2024-06-13 09:42:58	oth.eve.mdt.qq.com	10.255.35.159	bc:6e:e2:e1:70:da	10.3.9.5	00:0c:29:07:cb:ff	UDP	78	N/A	Technology/Internet
11	2024-06-13 09:42:58	oth.eve.mdt.qq.com	10.3.9.5	00:0c:29:07:cb:ff	10.255.35.159	bc:6e:e2:e1:70:...	UDP	142	0	Technology/Internet
12	2024-06-13 09:43:08	www.msftconnecttest.co...	10.255.35.159	bc:6e:e2:e1:70:da	10.3.9.5	00:0c:29:07:cb:ff	UDP	83	N/A	Technology/Software/Internet

- 使用OpenAI API对域名进行详细分析和解释，进行数据分析绘制动态饼图，进行各个域名大类和大类中的细分的直观显示。设计政治敏感、色情、暴力的域名分析会警告。



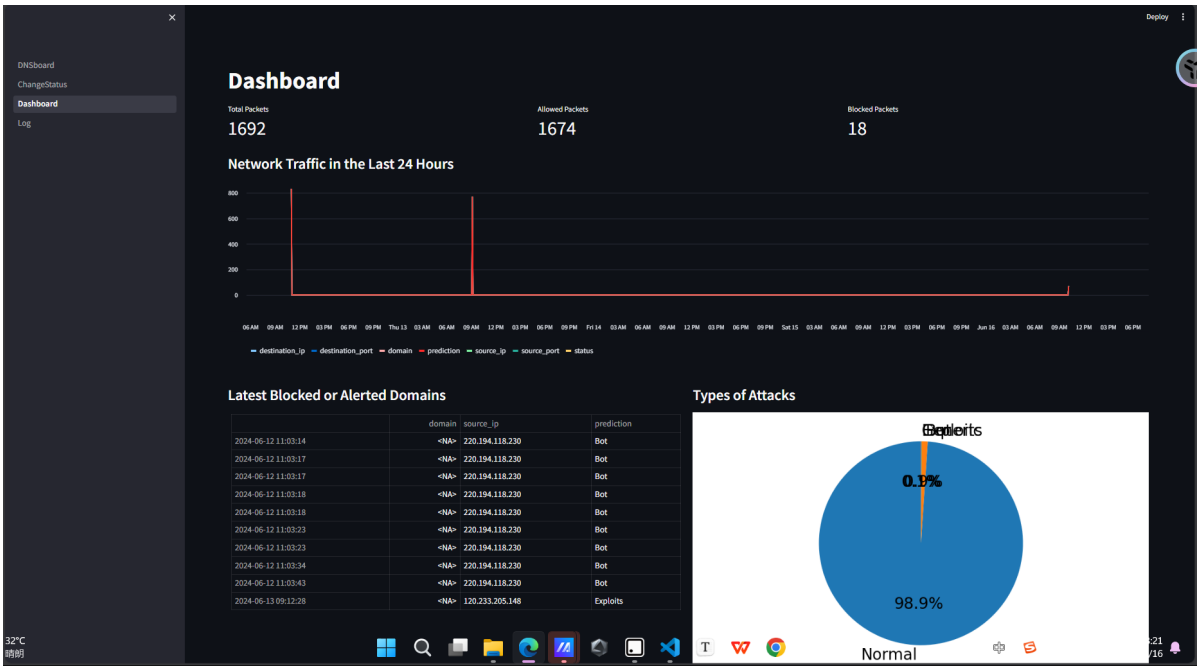


- 域名查询和分析功能，支持模糊查询，将查询到的结果发送给openai api，按固定的prompt进行分析。



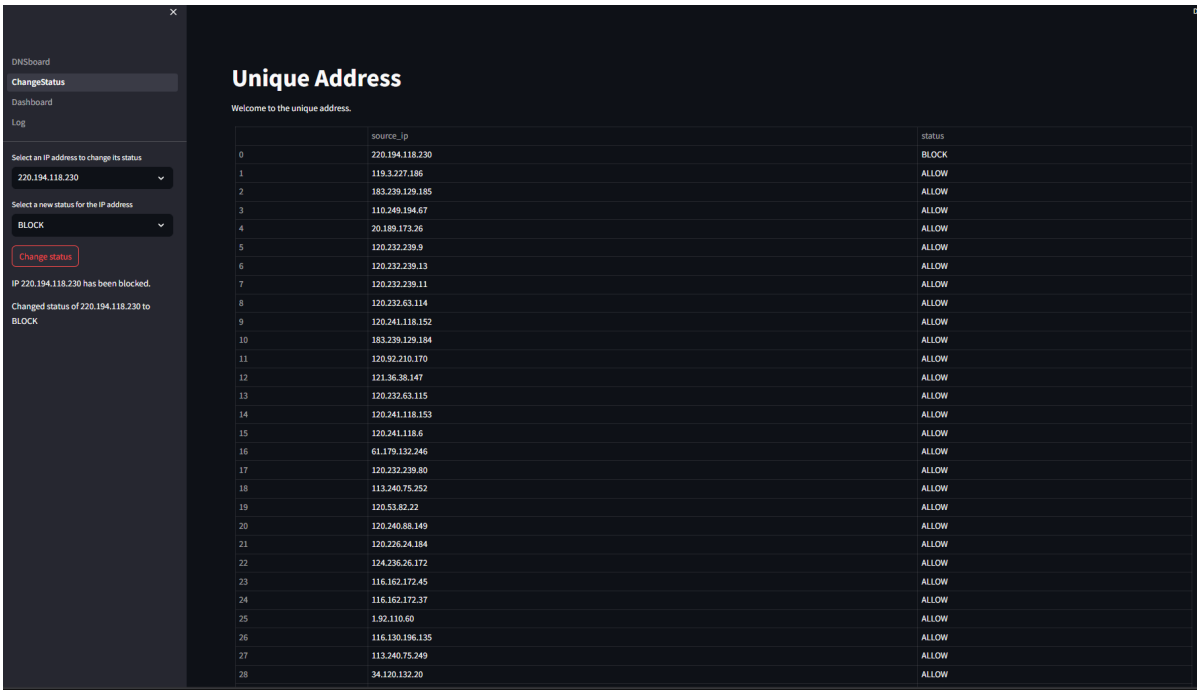
4.5.2 Dashboard

展示完整网络流量监控数据，标注异常与拥塞攻击流量



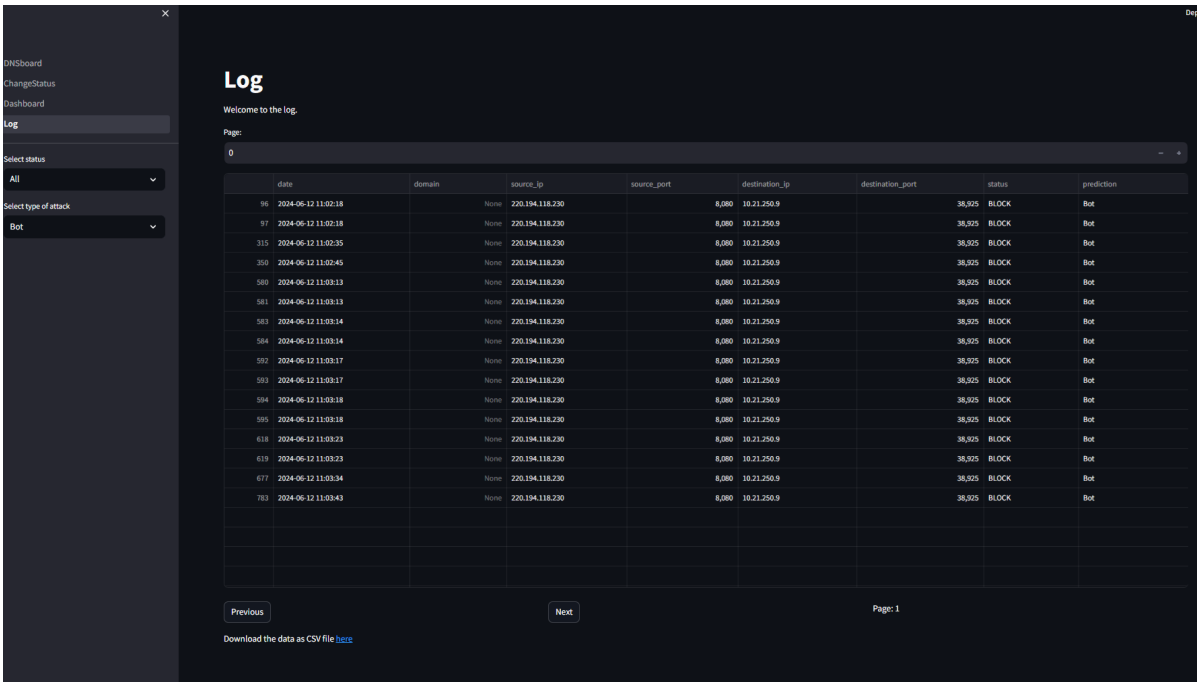
4.5.3 ChangeStatus

用户在Streamlit界面中选择一个IP地址并更改其状态时，系统会记录这一操作，并在日志中保存状态更改的时间和类型。实现了实时的网络流量控制，便于后续审计和分析。



4.5.4 Log

展示登陆日志，提供查询功能



五、 Summary

随着网络技术的不断发展和网络攻击手段的日益复杂，传统的网络流量监控技术已难以应对当前的安全挑战。通过将大模型技术，如深度学习和神经网络，应用于网络流量监控，我们能够显著提升网络安全的监测和防护能力。本文探讨了大模型技术在Linux后门检测、基于双向LSTM模型的网络攻击检测、以及网络入侵检测和监控工具的具体应用，展示了这些技术在提高检测准确性和响应速度方面的优势。

通过大模型技术的引入，网络流量监控不仅能够实现对海量数据的快速分析和异常检测，还能提供更加智能化和自动化的安全防护手段。同时，利用实时网络流量分析和监控工具，可以更有效地识别和阻止潜在的网络攻击，保障网络环境的安全与稳定。

未来，我们将进一步探索大模型技术在网络安全领域的应用，不断完善和优化现有技术手段，以应对不断变化的网络威胁。我们相信，随着技术的进步和应用的深入，网络安全将迈向一个更加智能和高效的新时代。

参考文献

1. Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*. MIT Press.
2. Chen, T., & Guestrin, C. (2016). XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*.
3. Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, 9(8), 1735-1780.
4. Yin, C., Zhu, Y., Fei, J., & He, X. (2017). A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks. *IEEE Access*, 5, 21954-21961.
5. Guo, H., & Li, Y. (2019). Network Traffic Analysis and Anomaly Detection Based on Convolutional Neural Networks. *IEEE Access*, 7, 10024-10031.
6. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.