UNIVERSITY OF LIMOGES
FACULTY OF SCIENCES AND TECHNIQUES

Master 1
Analyse et Développement Logiciel

Network Intrusion Detection System (NIDS) based on Deep Learning

NGUYEN Viet-Sang
NGUYEN Phuong-Hoa
TRAN Ngoc-Nhat-Huyen

Supervisor:
M. CONCHON Emmanuel, University Professor

April, 2020

Acknowledgments

We would like to wholeheartedly express the gratitude to our supervisor, M. CONCHON Emmanuel, for his support during the two semesters. His guidance helped us in all the time of research and implementation. We had a lot of difficulties in completing the project, but thanks to monthly meetings and his enthusiastic support of advice as well as test equipment, we overcame it all. Without his persistent help, the goal of this project would not have been realised.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

In recent years, the number of people using the Internet and network services is increasing day by day. Cyber-attacks are becoming more complex and noticeable. Network intrusion detection become more and more critical to defend information systems against network attacks. Previously NIDS was built on historical network data and system logs. Data Management Systems or Systems for Data at Rest have their own limitations while the explosion of data nowadays. Because they are expensive and not real-time in network environments, streaming systems have been developed to solve that problem. Besides, Artificial Intelligent (AI) provides a large number of applications which support for other fields including security.

1.1 Objectives

In this project, we aim to build a real-time system which is able to capture network intrusion and then predict its type of attack by an AI model. More specifically, the main goals are:

- Using an Intrusion Detection System (IDS) such as Snort or Suricata to capture network intrusion.
- Using the streaming system Kafka as a server to coordinate data with upstream and downstream.
- Analysing captured traffic by a network traffic analyser such as Zeek.
- Building a Deep Learning model to predict types of attack.
- Combining above separated parts to a real-time system.

Recall that in the previous semester, we have walked through some knowledge and demo needed for implementation. We studied the basic concepts of different streaming system and came to the chosen of Kafka and Spark as two parts in our general architecture. We also studied the method of building an AI model based on Deep Learning and practiced some examples. Moreover, the analysis and pre-processing of KDDCup99 dataset was done in last semester. Our tasks in this semester is to implement separated parts of the real-time system and combine them together. The video of demo can be found at https://www.youtube.com/watch?v=5G5mZ2JT_jo.

This report is organised as follows:

- Chapter 1 presents the objectives of this project. We then show the whole architecture of the real-time system and briefly describe it.
- Chapter 2 provides the details of implementation including 5 main parts. They are capturing network intrusion with Snort, coordinating data with Kafka server, transforming pcap to

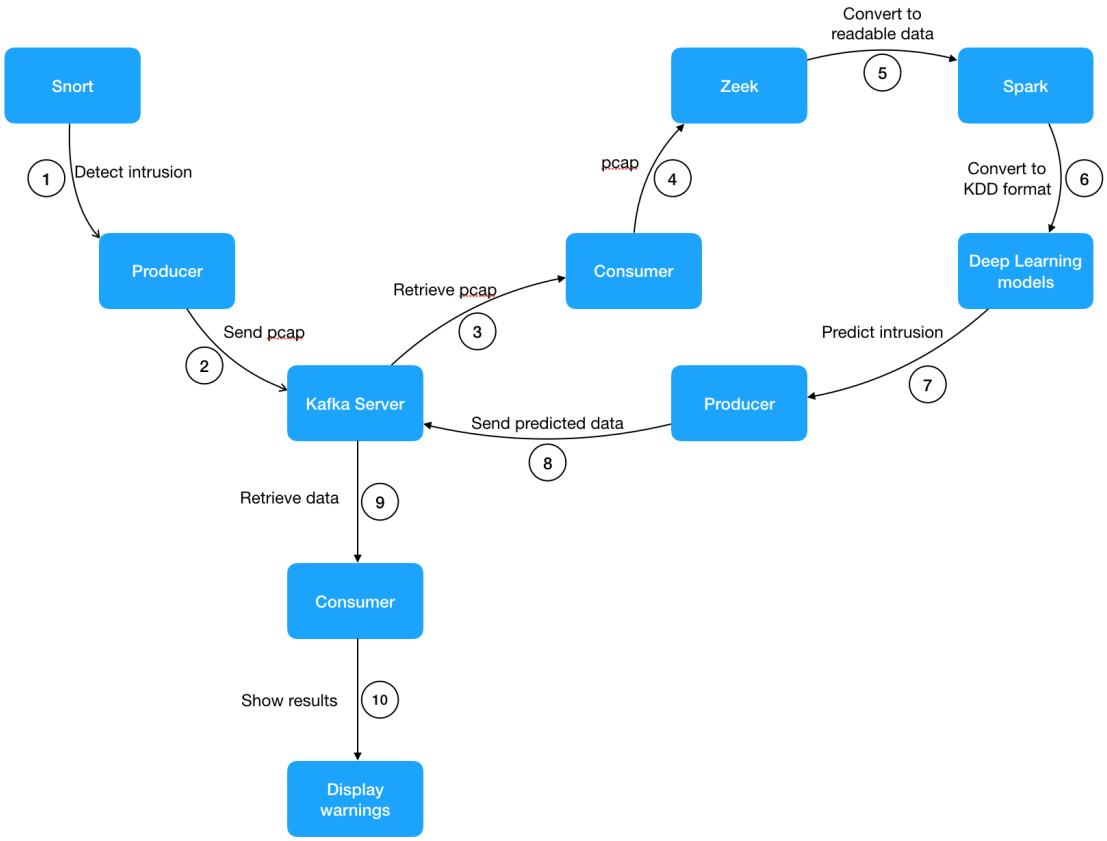


Figure 1.1: Architecture of system

readable logs with Zeek, derivation of logs in KDDCup99 format with Spark and finally, predicting type of attacks with AI model.

- Chapter 3 shows our results. We also discuss some drawbacks and propose some approaches to develop the system in this chapter. We then come to conclusions for this project.

1.2 Overview of system

In this section, we present the general architecture of our system. The goal of this system is to combine a NIDS with an AI model to increase detection and prediction rates. The NIDS firstly captures suspect traffic according to pre-defined rules. The AI model assists the NIDS in predicting attacks. This architecture is also used in many large scale IDS using Big Data technologies [3].

In our system, we use Snort[1] as an IDS to sniff and capture suspect packets. Kafka [2] plays a role as a streaming-processing platform. The basic concept behind Kafka in our architecture is to have *producers* and *consumers* to communicate with Kafka server. Producers publishes data to the server and consumers retrieves data from this server to process. We use Zeek[3] as a network optimized programming language to analyse *pcap* logs and transform them to readable data. Zeek (formerly Bro) is the world's leading platform for network security monitoring. Spark is utilised to convert data to KDDCup99 format which we have already presented in the previous report. Then, an AI model predicts that which type of attacks the data belongs to, or it is just normal data.

[1]https://www.snort.org
[2]https://kafka-python.readthedocs.io/en/master/
[3]https://zeek.org

Figure 1.1 shows the overview of the system. As we can see, there are 10 stages in this process. In chapter 2, we present our work in details corresponding to each stage. We briefly explain each stage as follows.

1. Snort captures suspect traffic based on pre-defined rules.
2. A producer sends suspect traffic to Kafka server.
3. A consumer retrieves data which is sent in stage 2.
4. The consumer waits for a batch of N (pre-defined argument) samples of data, then applies Zeek.
5. Zeek transforms data of pcap format to readable data. This readable data also has its own format to support for calculation of KDDCup99 data.
6. Spark converts data to KDDCup99 format.
7. AI model predicts type of attacks for data.
8. A producer sends predicted data back to Kafka server.
9. A consumer retrieves predicted data from Kafka server.
10. Display results and warn to users.

In the next chapter, we present the implementation corresponding to each stage in this architecture.

Chapter 2

Implementation details

In this chapter, we present the implementation in each stages shown in figure 1.1. More specifically, we discuss the way we use Snort to capture intrusion in subsection 2.1, then Kafka streaming system to coordinate data in subsection 2.2. The presentation of using Zeek to process pcap logs is subsection 2.3. Last but not least, application of an AI model is explained in subsection 2.5. To sharpen our system, we implement a program to display warnings which is shown in subsection 2.6. Normally, there are two main parts in each subsections. The first part is to describe and present how we propose problems, research and come to solutions. The second is to present how we adapt the solutions to the real-time system, that is, the connections between separated components.

2.1 Capturing suspect traffic

The first stage in our real-time architecture is to detect intrusion. The information of intrusion is logged and then sent to Kafka server thanks to a producer. This is considered as the first basic step to monitor our devices in network environment. In this section, we present two main parts, including how we use Snort and how we adapt it to the real-time system. That means, the work of stages 1 and 2 are discussed in this section.

Work of Snort

As mention earlier, we use Snort as an IDS to listen and capture suspect packets. To run Snort, there are two configurations that we should notice. The first one is to configure the file called *snort.config*. In this file, we must point to an another file which contains rules as shown in listing 2.1. This is a simple configuration. We could use a more complex configuration with several steps by modifying the template of Snort. However, in our system, we use this one for illustrative purpose. As we can see in listing 2.1, it points to the file *icmp.rules* in folder *rules*.

```
1 include rules/icmp.rules
```

Listing 2.1: Configuration of *snort.config*

We next discuss about rules. To capture suspect traffic, we must declare rules which could be understood as the answer for the question "What are suspect traffic ?". There are something that need to be declared such as source IP, destination IP, source port, destination port, protocol, etc. In our example, we define a simple rule to capture all ICMP packets as shown in listing 2.2. Let's walk through the syntax of this rule. There are two main parts in a Snort's rule including *header* and *options*. Header of our rule contains the following components:

- `alert`: **rule action**. Snort will generate an alert when the condition is met.

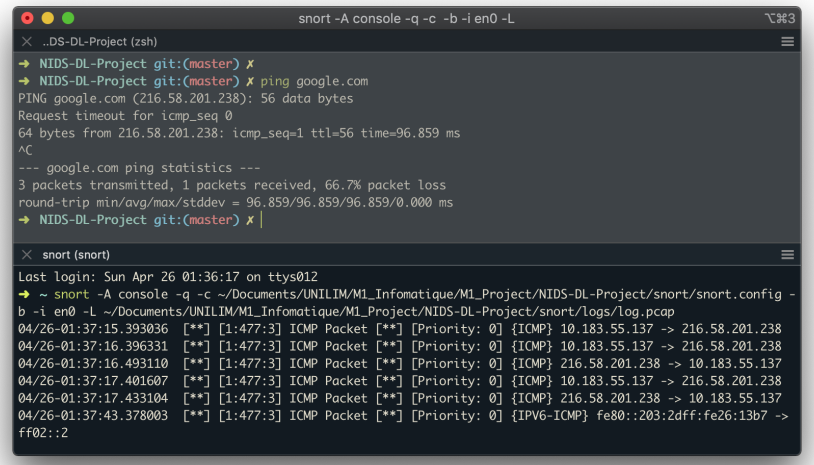


Figure 2.1: Snort captures ICMP packets

- `any`: **source IP**. Snort will look at all sources.
- `any`: **source port**. Snort will look at all ports.
- $->$: **direction**. It is from source to destination.
- `any`: **destination IP**. Snort will look at all destination.
- `any`: **destination port**. Snort will look at all ports.

```
alert icmp any any -> any any (msg:"ICMP Packet"; sid:477; rev:3;)
```

Listing 2.2: Configuration of *icmp.rules*

Options in a rule contains some components such as message along with the alert, ID of that rule. The meaning of options in the rule shown in listing 2.2 are as follows.

- `msg:"ICMP Packet"`: **message**. Snort will include this message with the alert.
- `sid:477`: **rule ID**
- `rev:3`: **revision number**. This option allows for easier rule maintenance

Run Snort: In the command of running Snort, we must specify some arguments. The three most important arguments are interface, the path to *snort.config* and the path to log file. The command as shown in listing 2.3 sniffs on interface *en0* (`-i en0`) with the configuration on *snort.config* (`-c /.../snort.config`) and writes logs to file *log.pcap* (`-L /.../log.pcap`). Besides, there are some other options. `-A console` is to show alerts on terminal and `-q` is to hide Snort's banner.

```
snort -A console -q -c ~/.../snort.config -i en0 -L ~/.../log.pcap
```

Listing 2.3: Command to run Snort

Figure 2.1 shows an example of running Snort. In this example, we run the command above and aim to capture ICMP packets. We then test by pinging to `www.google.com`. As we can see, Snort is able to capture all ICMP packets as expected and shows alerts to terminal.

Adapt to real-time system

Soon after Snort detects an intrusion, a log under format of pcap is generated. We immediately send this log to Kafka server by using a producer. The work of how the producer sends messages

to Kafka is presented in the next section. Up to this point, we have walked through stages 1 and 2 in the whole architecture.

2.2 Kafka Streaming System

Kafka server in our system plays a role as a data coordinator. It is a place where producers send data to and consumer retrieves data from. In this section, we present three main parts, including how to run a Kafka server, how to publish data to the server and how to retrieve data from it. These are the work of stages 2, 3, 8 and 9 in our general architecture.

Run Kafka server

To install Kafka, we simply download the binaries and run it. Apache Kafka is written in Scala, which compiles to Java bytecode. Therefore, Java SE Development Kits (JDK) need to be already installed on our system. It requires Java 7 or greater. We highly recommended to use Java 8.

We use curl to download the Kafka binaries:

```
$ curl http://apache.crihan.fr/dist/kafka/2.3.1/kafka_2.12-2.3.1.tgz -o ~/.../
    kafka_2.12-2.3.1.tgz
```

Kafka relies on Zookeeper. Firstly, we need start Zookeeper server which is a top-level software that keeps track status and store metadata of the Kafka cluster nodes and topics, partitions.

```
$ ~/[kafka dir]/zookeeper-server-start ~/[kafka dir]/config/zookeeper.properties
```

To start Kafka broker server, we open another command prompt and type this command:

```
$ ~/[kafka dir]/bin/Kafka-server-start.sh  ~/[kafka dir]/config/server.properties
```

Publish data (producer)

In order to write messages to Kafka, we create a producer object with the following properties:

- `bootstrap.servers` contains `host:port` pair, helps producer can find the Kafka cluster.
- `api_version`: specify which Kafka API version to use

```
producer = KafkaProducer(bootstrap_servers=['localhost:9092'], api_version=(0, 10))
```

Listing 2.4: Create a producer

Each log record is defined by a key-value pair and sent to Kafka topic. A topic is a category to which data records or messages are published. Here, all data streams which are in the same stage, will be assigned to the same topic. We use the producer object `send()` method to send logs.

```
def publish_message(producer, topic, key, value):
  try:
    #encode key and value to raw bytes which is what kafka stores
    key_bytes = bytes(key, encoding='utf-8')
    value_bytes = bytes(value, encoding='utf-8')
    producer.send(topic, key=key_bytes, value=value_bytes)
    producer.flush()
    print('Publish message number {}...'.format(key))
  except Exception as e:
    raise e
```

Listing 2.5: Producer publish message to Kafka server

Logs are read from file and published line by line. Key values are incremental variables coming with logs.

```
key = 0
while True:
  for file in os.listdir(PATH_TO_LOGS):
    f = open(os.path.join(PATH_TO_LOGS, file), 'rb')
    while True:
        line = f.readline()
        key += 1
        publish_message(producer, TOPIC_NAME, key, line)
```

Listing 2.6: Update changes of Snort and send data to Kafka Server

The system uses function of Kafka producer in the work of stages 2, 8 with different in parameters, especially the topic name. Kafka producer (stage 2) proceed to send pcap to Kafka server. The task of streaming is continuous until reaching 10 consecutive times of reading nothing. Another producer is in stage 8. It is responsible for sending the predicted data back to Kafka Server. Based on the work, we use topic *logfromsnort* for stage 2 and topic *warninglog* for stage 8.

Retrieve data (consumer)

Kafka consumer needs subscribe to the appropriate topic, and start receiving messages. The systems contains 2 consumers with different purposes. One that is in stages 3, receives messages which is stages 2. So, it has to subscribe topic *logfromsnort*. Then, this program is going to wait for a batch of N samples data to transport to Zeek.

There are some arguments to configure a consumer.

- `auto_offset_reset`: set to *'earliest'* to consume topic from start.
- `bootstrap_servers`: need to connect to Kafka (usually port 9092).
- `consumer_timeout_ms`: number of millisecond to throw a timeout exception to the consumer if no message is available for consumption.

```

consumer = KafkaConsumer(TOPIC_PCAP_LOG,
  auto_offset_reset='earliest',
  bootstrap_servers=['localhost:9092'],
  api_version=(0,10),
  consumer_timeout_ms=1000)

# Load tensorflow models
md1, md2, md3 = load_models()

batch_number = 0
sample_number = 0
while True:
  batch_number += 1
  print('Processing batch number {}...'.format(batch_number))
  batch = consumer.poll(TIMEOUT, BATCH_SIZE)
```

Listing 2.7: Consumer subscribes and retrieves message from Kafka server

In listing 2.7, at line 16, we call `consumer.poll(...)`, it returns a batch of records written to Kafka that consumers in our group have not read yet. The parameter we pass `TIMEOUT` that controls how long poll() will block if data is not available in the consumer buffer. Moreover, this configuration assists consumer to wait and retrieve more messages per iteration. Parameter `BATCH_SIZE` defines maximum size of batch of messages.

In stage 9, Another consumer subscribes and waits for results after prediction process. Its topic must be identical to topic name in stage 8, *warninglog*.

2.3 Convert pcap to readable logs (Zeek)

After retrieving data from Kafka server (stage 4), we have a batch of pcap logs. In practice, we set the number of sample in a batch to 50. It means consumer waits for a batch of 50 samples before going to the next step of processing. These samples are then processed by Zeek in order to transform them to readable data. After that, we convert the result to KDDCup99 format so as to adapt with AI model. In this section, we present two main parts, including how we use Zeek and how we adapt it to the real-time system. These are the work of stages 4 and 5 in general architecture.

Work of Zeek

We generate our database in the same format with KDDCup99 which is a broadly used database in the scientific environment. After capturing network traffic as a pcap file, contained all the packages transmitted in the simulated network, so we could identify a connection and compute its features. While some features can be easily extracted from the packet header, others must go through several computational steps. The KDDCup99 database has 41 predictor features divided in three groups:

- **Basic features:** the features obtained by analysing the *headers* of the network packages, such as duration, flag, service name, protocol, source bytes, destination bytes, etc. All these features are directly *available* in the header fields without doing any processing.
- **Content features:** the features obtained by analysing the *payloads* of the network packages, such as, if the root user has got the shell or not, number of failed login attempts, number of file creation operations, etc. These features are proposed by experts' using their knowledge in network security that look for suspicious behavior in the data portions. They are *not available* and need doing little processing.
- **Traffic features:** the features computed by analysing the *previous connections*, such as, the number of connections accessing to the same IP address in the last two seconds or between the last 100 connections, number of connections to the same service as the current connection in the past two seconds. Therefore, to calculate these features unlike the other previous, we need the information about a *group of connections* to be able to calculate them.

To extract basic features (IP addresses, port numbers, service names, flag, etc.), content features and payloads from tcpdump files, writing scripts that suits us, we used an available library based on Bro. *Bro-IDS*, recently renamed to Zeek, is a powerful software framework for analyzing traffic that is most commonly used to detect behavioral anomalies on a network for cyber-security purposes. Zeek works on most modern, Unix-based systems and requires no custom hardware. Zeek scripts can be identified by the use of a `.bro` or `.zeek` file name extension. In `darpa2gurekddcup.bro` script file, we get directly available **basic features**, and compute **content features**.

```
zeek -r log.pcap darpa2gurekddcup.bro > log.list
# bro -r log.pcap darpa2gurekddcup.bro > log.list
```

Listing 2.8: Command to run Zeek (Bro)

The command of running Zeek as shown in listing 2.8. This command tells Zeek which script to load. Zeek will output log files into the working directory. Zeek extracts and processes two types of features from `log.pcap` input file then writes logs to file `log.list`.

We now discuss how Zeek operates to extract the basic features. IP packets captured with pcap are transferred to an event engine of Zeek to process by the policy script interpreter. The event engine analyzes network traffic to generate events when "something" happens. Events are actually a special flavor of functions but are essential to how Zeek works. In our program, we use some events defined by Zeek for the primary parts of any connection such as `event packet_content(...)`

shown in listing 2.9. This event is generated for every packet that has a non-empty transport-layer payload.

Consider the example in listing 2.9, we firstly define a schema of data which we would like in the output (lines 3-16). `num_conn` (line 1) is a variable considered as an identify number of each sample. We create a new table of data (`connection_tcp`) of the above schema. Data in this table has primary keys which are tuple of a string, two IP addresses and two ports (`table[string, addr, port, addr, port]`). We notice that `string`, `addr`, `port` are primitive types in Zeek language. Our idea is to use a tuple of timestamp, origin host (`orig_h`), origin port (`orig_p`), destination host or response host (`resp_h`) and response port (`resp_p`) as a primary key for each sample. Whenever Zeek process a sample of pcap, it automatically run the event `packet_contents`. It also extracts information of `connection` and provides us to do what we would like inside this event. For instance, line 22 gets the value of `orig_h` (type of `addr`) by the command `c$id$orig_h`. This command is similar to `c.id.orig_h` in Python or many other languages, but in a different syntax because of dots in any IP address. Line 27 assigns a value to the field service in data table. This value is the returned result of funtion `service_name(...)`.

```
1  global num_conn=0;
2
3  type schema: record {
4    num_conn : count &default = 0;
5    orig_h : string;
6    resp_h : string;
7    orig_p : string;
8    resp_p : string;
9    duration : string &default = "0";
10   protocol : string &default = "tcp";
11   service : string &default = "";
12   flag : string &default = "OTH";
13   src_bytes : string &default = "0";
14   dst_bytes : string &default = "0";
15  # ...
16 };
17
18 global connection_tcp: table[string, addr, port, addr, port] of schema;
19 # table[timestamp, orig_h, orig_p, resp_h, resp_p]
20
21 event packet_contents(c: connection, contents: string){
22   local orig_h : addr = c$id$orig_h;
23   local resp_h : addr = c$id$resp_h;
24   local orig_p : port = c$id$orig_p;
25   local resp_p : port = c$id$resp_p;
26   # ...
27   connection_tcp[startTime, orig_h, orig_p, resp_h, resp_p]$service = service_name(
       resp_p);
28   # ...
29 }
```

Listing 2.9: Events of Zeek

We next consider the implementation of extracting service as shown in listing 2.10. Recall that when calling the function in event `packet_contents` (line 27 in listing 2.9), we pass `resp_p` as a parameter. If we know the port number of the destination host, we can find the name of service by using the `service_name(p:port):string` function from the Zeek script. For example, some service such as `ftp`, `ssh`, `smtp`, `domain` runs on port 21, 22, 25 and 53, respectively. Listing 2.10 shows `service_name` function.

```
1 function service_name(p: port): string
2 {
3     if (p==21/tcp)       return "ftp";
4     else if (p==22/tcp)  return "ssh";
5     else if (p==25/tcp)  return "smtp";
6     else if (p==53/tcp)  return "domain";
7     else if (p==80/tcp || p==8008/tcp || p==8080/tcp)  return "http";
8     # ...
9     else return "other";
```

```
}
```

Listing 2.10: Zeek script to extract service feature

Similarly to extract `service`, we do the same for others basic features. We take another example of content features, feature `num_root`, which is the number of operations done in root mode in a connection. Function `num_root(connection, user)` is shown in listing 2.11. In the content area of the network packages, we just need to check if user is root or not then increase the counter variable by 1.

```
function num_root(c: connection, user: string){
    local trans : string = fmt("%s", get_port_transport_proto(c$id$resp_p));
    local timestamp : string = fmt("%.6f", c$start_time);
    local duration : string = fmt("%.6f", c$duration);
    local orig_h : addr = c$id$orig_h;
    local resp_h : addr = c$id$resp_h;
    local orig_p : port = c$id$orig_p;
    local resp_p : port = c$id$resp_p;

    if(/[rR][oO][oO][tT]/ in user){
        if(trans=="tcp")
            ++ connection\_tcp[timestamp,orig_h,orig_p,resp_h,resp_p]$num_root;
        else
            ++ connection[timestamp,duration,orig_h,orig_p,resp_h,resp_p]$num_root;
            # table data of UDP and ICMP
    }
}
```

Listing 2.11: Zeek script to compute num_root feature

There are 19 **traffic features** and they are computed using the connections preceding the current connection. We divide them into two groups: (1) time traffic features and (2) machine traffic features. The first group is computed previous connections with fixed time-window (in the last 2 seconds). And the second group is for fixed number of connections (in the last 100 connections). In each group, we calculate the number of connections or the percentage of connections that satisfied a certain condition.

For example, we create `srv_count` feature, which is the number of connections to the same destination port number as the current connection (in the last 2 seconds) as shown in listing 2.12. `line1` is the current connection, we travel all previous connections (represented by `line2`). `resp_p1, resp_p2` are the corresponding destination port. On lines 11-12, we check if these two connections are in the last 2 seconds and have the same destination port, then increase `srv_count` variable by 1 (line 13).

```
for i, line1 in enumerate(content):
    srv_count = 0
    start_time1 = line1[1]
    resp_p1 = line1[3]
    start_time_11, start_time_12 = start_time1.split('.')
    for j in range(i):
        line2 = content[j]
        start_time2 = line2[1]
        resp_p2 = line2[3]
        start_time_21, start_time_22 = start_time2.split('.')
        if start_time_11 - start_time_21 <= 2 and start_time_21 <= start_time_11:
            if resp_p1 == resp_p2:
                srv_count += 1
```

Listing 2.12: Calculate srv_count feature

Listing 2.13 shows an example for creating a machine traffic feature. Feature `dst_srv_host_count` is a number of connections to the same destination port number as the current connection's (in the last 100 connections). Similarly to time traffic feature, we travel all previous connections and count the number of connections that have same destination port.

```python
for i, line1 in enumerate(content):
    dst_srv_host_count = 0
    resp_p1 = line1[3]

    k = 0 if i <= 100 else i - 100
    for l in range(k, i):
        line3 = content[l]
        resp_p3 = line3[3]
        if resp_p1 == resp_p3:
            dst_srv_host_count += 1
```

Listing 2.13: Calculate feature `dst_srv_host_count`

Finally, we run `convert.py` Python program to create **traffic features** as shown in listing 2.14. In this Python command, we must specify input and output file in arguments. The logs are available in `fake_KDD.txt` output file with 41 features. Figure 2.2 shows an example for logs in KDDCup99 format which are outputs of this part.

```
python3 convert.py --in_file sorted.list --out_file fake_KDD.txt
```

Listing 2.14: Command to create traffic features

```
100.077598 tcp http_443 SF 1 58 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
277.127861 tcp http_443 OTH 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 1 0 0 0.0 1.0 0 0.0 0 0.0
240.035618 tcp http_443 SF 776 1023 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0.0 0 0.0 0 0 1.0 0 2 0 0 0.0 1.0 0 0.0 0 0.0
274.691053 tcp http_443 OTH 572 1322 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0.0 0 0.0 0 0 1.0 0 3 0 0 0.0 1.0 0 0.0 0 0.0
218.529846 tcp http_443 SF 1 63 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 4 0 0 0.0 1.0 0 0.0 0 0.0
100.292046 tcp http_443 SF 1 58 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 5 0 0 0.0 1.0 0 0.0 0 0.0
58.545163 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 6 0 0 0.0 1.0 0 0.0 0 0.0
0.015698 tcp http RSTRH 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.0 0 0 0 0 0 0
58.445904 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 7 0 0 0.0 1.0 0 0.0 0 0.0
242.126536 tcp http_443 OTH 23580 4135 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 8 0 0 0.0 1.0 0 0.0 0 0.0
103.836309 tcp http RSTRH 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0.0 1.0 0 0.0 0 0.0
104.204414 tcp http RSTRH 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 2 0 0 0.0 1.0 0 0.0 0 0.0
243.027619 tcp http_443 OTH 280 240 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 9 0 0 0.0 1.0 0 0.0 0 0.0
58.334733 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 10 0 0 0.0 1.0 0 0.0 0 0.0
218.436130 tcp other OTH 27 26 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.0 0 0 0 0 0 0
70.322470 tcp http_443 SF 1 58 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 11 0 0 0.0 1.0 0 0.0 0 0.0
221.200739 tcp http_443 OTH 1843 1214 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 12 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
10.205322 tcp http_443 SF 1 81 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0.0 0 0.0 0 0 1.0 0 13 0 0 0.0 1.0 0 0.0 0 0.0
231.232411 tcp http_443 OTH 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0.0 0 0.0 0 0 1.0 0 14 0 0 0.0 1.0 0 0.0 0 0.0
88.945253 tcp http_443 SF 1 31 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0.0 0 0.0 0 0 1.0 0 15 0 0 0.0 1.0 0 0.0 0 0.0
28.372787 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0.0 0 0.0 0 0 1.0 0 16 0 0 0.0 1.0 0 0.0 0 0.0
209.380365 tcp http_443 OTH 8017 963 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0.0 0 0.0 0 0 1.0 1 17 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
188.191993 tcp http_443 SF 1 63 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0.0 0 0.0 0 0 1.0 0 18 0 0 0.0 1.0 0 0.0 0 0.0
220.153472 tcp http_443 OTH 290 235 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 5 0 0.0 0 0.0 0 0 1.0 0 19 0 0 0.0 1.0 0 0.0 0 0.0
13.274497 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 6 0 0.0 0 0.0 0 0 1.0 1 20 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
13.179338 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0.0 0 0.0 0 0 1.0 0 21 0 0 0.0 1.0 0 0.0 0 0.0
13.173278 tcp http_443 SF 1 56 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0.0 0 0.0 0 0 1.0 0 22 0 0 0.0 1.0 0 0.0 0 0.0
54.776087 tcp http_443 SF 1 58 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 2 0 0.0 0 0.0 0 0 1.0 0 23 0 0 0.0 1.0 0 0.0 0 0.0
198.807658 tcp http_443 OTH 65949 7140 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 3 0 0.0 0 0.0 0 0 1.0 1 24 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
213.704038 tcp http_443 OTH 499 177 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 4 0 0.0 0 0.0 0 0 1.0 1 25 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
216.834118 tcp http_443 S1 1777 2103 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 26 0 0 0.0 1.0 0 0.0 0 0.0
0.000000 udp other S0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0.0 0 0 0 0 0 0
```

Figure 2.2: Logs in `fake_KDD.txt` file

Adapt to real-time system

In short, Zeek transforms pcap logs to basic features and content features. We then use Python to derive traffic features. However, in the real-time system, we only adapt the work of Zeek. The derivation of Python program is transferred to Spark. We write Python program only for research purpose before adapting the methods of computation to Spark. We present how Spark does in the next section.

2.4 Derive logs of KDDCup99 format (Spark)

In section 2.3, after transforming pcap logs to readable data by Zeek, we also write a Python program to derive data in KDDCup99 format. However, in reality, the number of data that need to be processed at a certain moment is enormous. That is why we should use a batch processing system such as Spark [5] rather than sequential processing, i.e. Python program. In this section,

we present two main parts. The first part is to explain how we utilise Spark to derive data in KDDCup99 format from the results of Zeek. The second one is to present how we apply the work of Spark to the real-time system. This section describes the work of stage 6 in the whole architecture.

Work of Spark

Firstly, we build a schema of dataframe as shown in listing 2.15 to load data from previous step. There are 28 fields in this schema which are the results of Zeek's process. From values of theses fields, we aim to derive 19 *traffic features* as mentioned in section 2.3 by some computations. We notice that it is necessary to cast some fields of `StringType` to `IntegerType` or `FloatType` before calculating.

```
schema = StructType([
    StructField('num_conn', StringType(), True),
    StructField('start_time', StringType(), True),
    StructField('orig_p', StringType(), True),
    StructField('resp_p', StringType(), True),
    StructField('orig_h', StringType(), True),
    StructField('resp_h', StringType(), True),
    StructField('duration', StringType(), True),
    StructField('protocol', StringType(), True),
    StructField('service', StringType(), True),
    StructField('flag', StringType(), True),
    StructField('src_bytes', StringType(), True),
    StructField('dst_bytes', StringType(), True),
    StructField('land', StringType(), True),
    StructField('wrong_fragment', StringType(), True),
    StructField('urg', StringType(), True),
    StructField('hot', StringType(), True),
    StructField('num_failed_logins', StringType(), True),
    StructField('logged_in', StringType(), True),
    StructField('num_compromised', StringType(), True),
    StructField('root_shell', StringType(), True),
    StructField('su_attempted', StringType(), True),
    StructField('num_root', StringType(), True),
    StructField('num_file_creations', StringType(), True),
    StructField('num_shells', StringType(), True),
    StructField('num_access_files', StringType(), True),
    StructField('num_outbound_cmds', StringType(), True),
    StructField('is_hot_login', StringType(), True),
    StructField('is_guest_login', StringType(), True)
])

rdd = spark.sparkContext.parallelize(data)
df = spark.createDataFrame(rdd, schema)
```

Listing 2.15: Schema of dataframe

For example, we calculate new values called `rerror_rate` for each sample of data in dataframe `df`. This feature represents the percentage of connections that have "REJ' errors [7]. A supportive feature in this calculation is `count` which stands for number of connections to the same host as the current connection in the past 2 seconds. In the example shown in listing 2.16, we aim to calculate `rerror_rate` for a certain `row` of data in dataframe `df`. Lines 1-2 initialise the values of `count` and `rerror`. We apply the window of 2 seconds to filter the dataframe by a condition (lines 4-5). To calculate `count` (line 6), we filter samples which have the same destination with currently considered sample (`row.resp_h == col('resp_h')`). Similarly, we calculate `rerror` by filtering as line 7. Finally, `rerror_rate` is derived by the formula $rerror\_rate = rerror/count$ (lines 9-10).

```
count = 0
rerror = 0

ndf = df.filter((float(row.start_time) - col('start_time') <= 2) \
```

```
                & (col('start_time') <= float(row.start_time)))
count = ndf.filter(row.resp_h == col('resp_h')).count()
rerror = ndf.filter((row.resp_h == col('resp_h')) & (col('flag') == 'REJ')).count()

if count != 0:
  rerror_rate = rerror / count
```

Listing 2.16: Calculation of `rerror_rate`

In a similar way, we calculate values for 19 traffic features as shown in listing 2.17 (lines 3-21) by applying suitable conditions of filtering. Theses new features are constructed as a new schema named `nddf`. After calculations, we join the old schema `df` with this one `nddf` with the primary key `num_conn`. Then, we select 41 needed features from the the result of joining (line 26).

```
derived_schema = StructType([
  StructField('num_conn', IntegerType(), True),
  StructField('counte', IntegerType(), True),
  StructField('srv_count', IntegerType(), True),
  StructField('serror_rate', FloatType(), True),
  StructField('srv_serror_rate', FloatType(), True),
  StructField('rerror_rate', FloatType(), True),
  StructField('srv_rerror_rate', FloatType(), True),
  StructField('same_srv_rate', FloatType(), True),
  StructField('diff_srv_rate', FloatType(), True),
  StructField('srv_diff_host_rate', FloatType(), True),
  StructField('dst_host_count', IntegerType(), True),
  StructField('dst_srv_host_count', IntegerType(), True),
  StructField('dst_host_same_srv_rate', FloatType(), True),
  StructField('dst_host_diff_srv_rate', FloatType(), True),
  StructField('dst_host_same_src_port_rate', FloatType(), True),
  StructField('dst_host_srv_diff_host_rate', FloatType(), True),
  StructField('dst_host_serror_rate', FloatType(), True),
  StructField('dst_host_srv_serror_rate', FloatType(), True),
  StructField('dst_host_rerror_rate', FloatType(), True),
  StructField('dst_host_srv_rerror_rate', FloatType(), True)
])

derived_rdd = spark.sparkContext.parallelize(derived_data)
nddf = spark.createDataFrame(derived_rdd, derived_schema)
res_df = df.join(nddf, df.num_conn == nddf.num_conn).select(...)
```

Listing 2.17: Schema of derived fields

Adapt to the real-time system

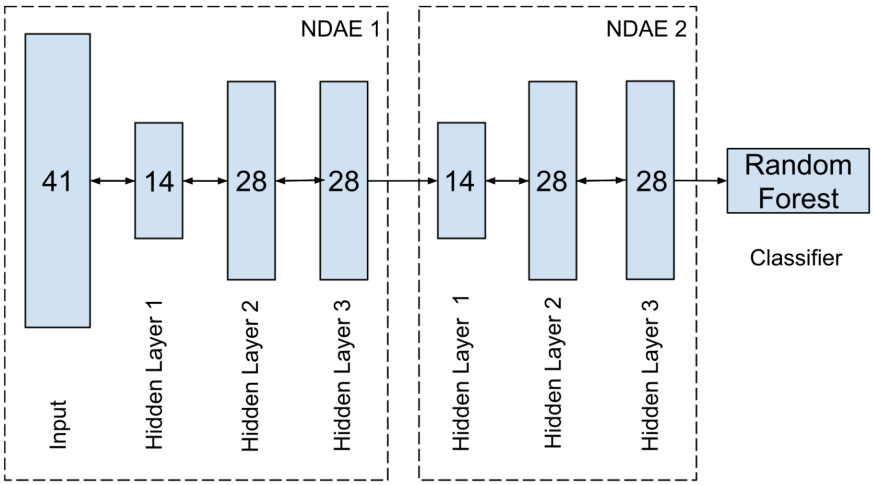
In the system, we define a function to do the work of Spark. After receiving a batch of data from Zeek, this stage continue processing to convert them to KDDCup99 format. The output of this stage is the data that can be fed to AI model. In our implementation, it is so slow for pesonal computers to run Spark. More specifically, it takes about 10 minutes to process a batch of 50 samples. However, this is only for illustrative purpose. In reality, people usually utilise a more powerful server (e.g. GPU) to run. In summary, we have presented the implementation of stages 6 in general architecture.

2.5 Predict type of attacks (AI model)

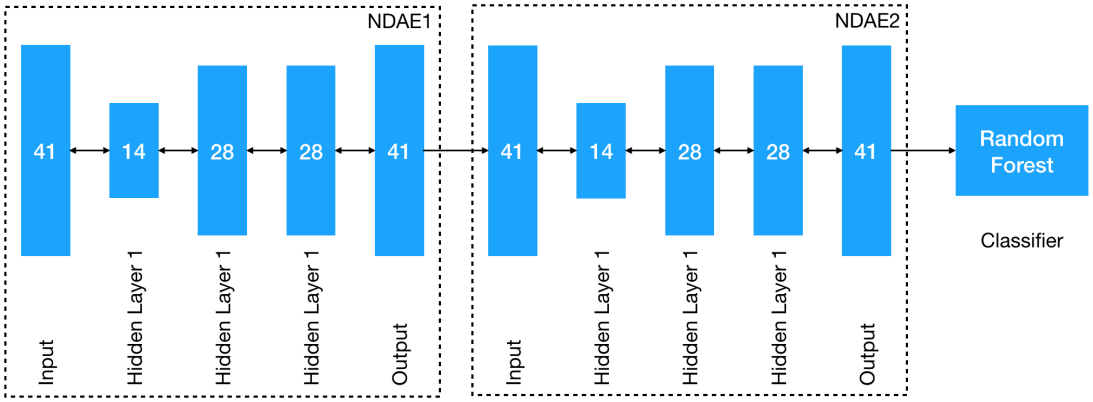
The contribution of AI in the general architecture we presented earlier is due to a pre-trained model. In the flow of our real-time system, it is only necessary to load this model and feed data to it in order to get the predicted result. To do that, we firstly obtain a model which is good enough to predict. Hence, we pose two problems in this stage including building and training a Deep Learning model based on the paper [2] and applying the pre-trained model to the real-time system. In this section, we present two main parts. The first one is implementing a neural network

model and training with KDDCup99 data [7]. And the second one is adapting the model to our real-time system.

Build a Deep Learning model



(a) Original model



(b) Our model

Figure 2.3: Changes in Deep Learning model

First of all, we discuss the architecture of model. Recall that our aim is to build a neuron networks model called "Stacked NDAE Classification Model" to learn the way to identify an attack from KDDCup99 data. 41 fields in this data are used as the inputs of model. The output is the prediction of attack type that a sample of data belong to among 5 possible types (Dos, Probe, U2R, R2L, Normal). In the previous report, we analysed the details of the model proposed in [2] (see figure 2.3a). However, there are some ambiguity in this architecture when comparing to the concepts relating to "Non-Symmetric Deep Auto-Encoder (NDAE)" (also provided in [2]). Therefore, we modify some this architecture based on the old.

The ambiguity in original model is that there is no output layer in each NDAE (figure 2.3a). If we accept that hidden layer 3 is the output layer, it is contradictory to the concept of auto-encoder that an auto-encoder is an unsupervised neural network-based feature extraction algorithm, which learns the best parameters required to **reconstruct its output as close to its input as possible**. It means the dimension of output layer must be equivalent to the dimension of input layer.

Moreover, if they are not equal, we do not have the labels of data to train since the labels are input data themselves. For those reasons, we modify the original model by add an output layer which has the same dimension as the input layer (see figure 2.3b).

We now show the way to implement the model with TensorFlow. Recall that the data is ready for training thanks to our work in the semester 1. The implementation of building and training model is shown in listing 2.18. Lines 1-4 load training data and testing data from *pickle* files which are saved in the data preparation stage. `train_x` contains values of 41 features while `train_y` contains type of attack of corresponding samples (labels). We notice that using *pickle* in Python 3 to save and load data is very convenient because it is unnecessary to reconstruct format of data again when loading. Lines 6-7 and 25-26 are to define a callback aiming to log the loss and accuracy when training. We will show this log and analyse it soon after.

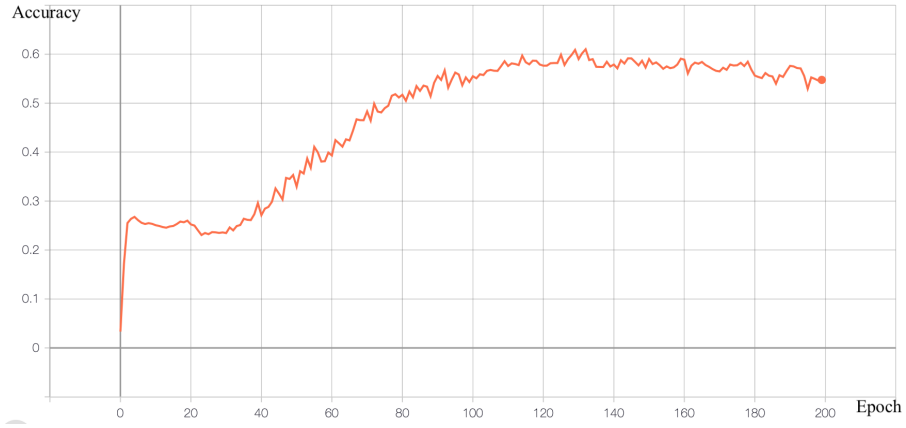
We next implement two NDAE blocks shown in figure 2.3b on lines 10-14 and 28-32. These two block are similar. We consider one of the two, for example, block NDAE 1 from line 10 to 14. We notice that `input_dim` on line 9 is equivalent to 41 which is the dimension of input layer (line 10) as well as output layer (line 14). Lines 11, 12, 13 are three hidden layers. As we can see, the output of previous layer is the input of the current layer. Line 16 defines the input and the output of model. We choose the types of functions for optimizer, calculation of loss and accuracy on lines 17-20. Training data is fed to model on line 21. Number of epochs and size of batch are set to 200 and 64, respectively. These arguments are large number because this dataset is a little bit small (about 126,000 samples). Moreover, it is not necessary to utilise GPU to train with this dataset.

```
1  file_train_x = open("data/train_x.pickle", "rb")
2  file_train_y = open("data/train_y.pickle", "rb")
3  train_x = pickle.load(file_train_x)
4  train_y = pickle.load(file_train_y)
5
6  log_dir_1="logs/m1" # Logs NDAE 1
7  tensorboard_callback_1 = tf.keras.callbacks.TensorBoard(log_dir=log_dir_1,
       histogram_freq=1)
8
9  input_dim = train_x.shape[1] # NDAE 1
10 in_data_1 = tf.keras.Input(shape=(input_dim,))
11 encoded_1 = tf.keras.layers.Dense(14, activation='relu')(in_data_1)
12 encoded_1 = tf.keras.layers.Dense(28, activation='relu')(encoded_1)
13 encoded_1 = tf.keras.layers.Dense(28, activation='relu')(encoded_1)
14 out_layer_1 = tf.keras.layers.Dense(input_dim, activation='sigmoid')(encoded_1)
15
16 model_1 = tf.keras.Model(inputs=in_data_1, outputs=out_layer_1)
17 model_1.compile(optimizer='adam',
18     loss='categorical_crossentropy',
19     metrics=['accuracy']
20 )
21 model_1.fit(train_x, train_x, epochs=200, batch_size=64, shuffle=True, callbacks=[
       tensorboard_callback_1])
22 model_1.save('models/model_1.h5')
23 out_1 = model_1.predict(train_x)
24
25 log_dir_2="logs/m2" # Logs NDAE 2
26 tensorboard_callback_2 = tf.keras.callbacks.TensorBoard(log_dir=log_dir_2,
       histogram_freq=1)
27
28 in_data_2 = tf.keras.Input(shape=(input_dim,)) # NDAE 2
29 encoded_2 = tf.keras.layers.Dense(14, activation='relu')(in_data_2)
30 encoded_2 = tf.keras.layers.Dense(28, activation='relu')(encoded_2)
31 encoded_2 = tf.keras.layers.Dense(28, activation='relu')(encoded_2)
32 out_layer_2 = tf.keras.layers.Dense(input_dim, activation='sigmoid')(encoded_2)
33
34 model_2 = tf.keras.Model(inputs=in_data_2, outputs=out_layer_2)
35 model_2.compile(optimizer='adam',
36     loss='categorical_crossentropy',
37     metrics=['accuracy']
38 )
39 model_2.fit(out_1, train_x, epochs=200, batch_size=64, shuffle=True, callbacks=[
```

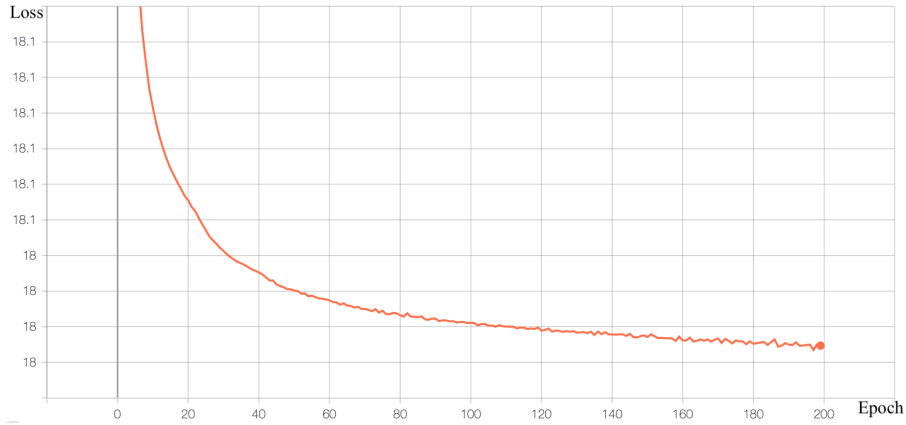
```
    tensorboard_callback_2])
model_2.save('models/model_2.h5')
out_2 = model_2.predict(out_1)

print("Training with Random Forest...") # Random Forest
parameters = {'bootstrap': True,
              'min_samples_leaf': 3,
              'n_estimators': 100,
              'min_samples_split': 10,
              'max_features': 'sqrt',
              'max_depth': 20,
              'max_leaf_nodes': None}
rf_model = RandomForestClassifier(**parameters)
rf_model.fit(out_2, train_y)
pickle.dump(rf_model, open('models/rf_model.sav', 'wb'))
```

Listing 2.18: Implementation of training model

We train two NDAE blocks separately and the output of the first one is the input of the second. Lines 43-53 show the implementation of Random Forest part. We setup some parameters for decision trees in Random Forest such as the maximum depth (`max_depth`) and number of trees (`n_estimators`). The output of block NDAE 2 is used as the input of Random Forest (line 52). After successfully training the model, we save it to files (lines 22, 40 and 53) and then test it.



(a) Accuracy in training



(b) Loss in training

Figure 2.4: Accuracy and loss in training

Figure 2.4 shows the accuracy and the loss when training with the block NDAE 1. These logs are recorded after each epoch of training. TensorFlow automatically divides the fed data into a training set and an another set called validation. After an epoch, the model is validated with the

validation set. The accuracy of this validation is shown in figure 2.4a. As we can see, the training process seems to be good from the beginning to epoch 140 since the accuracy gradually increases to around 60%. After that, the accuracy tends to decrease and hence, we stop training at that time. Meanwhile, the line of loss in figure 2.4b decreases over time. The shape of this line is what we expect when training any Deep Learning model. However, the decreasing range is so small (from 18.1 to 18). We could say that the accuracy as well as the loss in training are converging and the model is trained enough.

We next discuss how to test the pre-trained model with the code shown in listing 2.19. Lines 23-28 load the model and testing data from files. `test_x` contains 41 features of input and `test_y` contains labels corresponding to each sample. NDAE 1, NDAE 2 and Random Forest sequentially predict their results at the outputs (lines 30-32). The final prediction of Random Forest gives us the types of attack (`predictions`) which are used to compare with `test_y`. We define a function named `evaluate(predictions, labels)` on lines 1-12 to calculate the metrics mentioned in [2]. These metrics are `accuracy`, `precision`, `recall`, `false_adam` and `f1_score` based on concepts of True Positive (TP), False Positive (FP), True Negative (TN) and False Negative (FN).

```
1  def evaluate(predictions, labels):
2    TP = 0; FP = 0; TN = 0; FN = 0;
3
4    for pair in zip(predictions, labels):
5      if pair[0] != 0 and pair[1] != 0:
6        TP += 1
7      if pair[0] != 0 and pair[1] == 0:
8        FP += 1
9      if pair[0] == 0 and pair[1] == 0:
10       TN += 1
11     if pair[0] == 0 and pair[1] != 0:
12       FN += 1
13
14   accuracy = (TP + TN)/(TP + TN + FP + FN)
15   precision = TP/(TP + FP)
16   recall = TP/(TP + FN)
17   false_adam = FP/(FP + TN)
18   f1 = 2 * precision * recall / (precision + recall)
19
20   return accuracy, precision, recall, false_adam, f1
21
22 if __name__ == '__main__':
23   model_1 = tf.keras.models.load_model('models/model_1.h5')
24   model_2 = tf.keras.models.load_model('models/model_2.h5')
25   rf_model = pickle.load(open('models/rf_model.sav', 'rb'))
26
27   test_x = pickle.load(open('data/test_x.pickle', 'rb'))
28   test_y = pickle.load(open('data/test_y.pickle', 'rb'))
29
30   predictions = model_1.predict(test_x)
31   predictions = model_2.predict(predictions)
32   predictions = rf_model.predict(predictions)
33
34   accuracy, precision, recall, false_adam, f1 = evaluate(predictions, test_y)
35   print("accuracy = ", accuracy)
36   print("precision = ", precision)
37   print("recall = ", recall)
38   print("false_adam = ", false_adam)
39   print("f1 = ", f1)
```

Listing 2.19: Implementation of testing

Table 2.1 shows the results when testing and we also compare them to the original model's presented in [2]. At the first glance, we can see that our results are not as good as the ones in [2]. Accuracy, recall and f1-score of our model are 25-35% lower than the original. Precision is the most closet metric to the authors of [2]'s results (92.33% comparing to 99.99%). Our false Adam is also 4.58% higher than the one of old model. This performance of our model is not impressive because of several reasons. The first reason is the modification of the model's architecture. As we discuss earlier, we add an output layer to each NDAE block after carefully analysing. The

	Original model	Our model
Accuracy (%)	97.85	75.09
Precision (%)	99.99	92.33
Recall (%)	97.85	61.34
F-score (%)	98.15	73.71
False Adam (%)	2.15	6.73

Table 2.1: Comparison of training results

second reason is the authors of [2] did not provide the details of implementation. Some important arguments of a TensorFlow model such as size of batch or number of epochs are not provided and hence, we define them ourselves based on our experience. The third reason is that the authors did not present the pre-processing data stage. That stage plays a crucial role when working with any Deep Learning model. In summary, it is not fair to compare our model with the original because of many different conditions.

Adapt the model to real-time system

Despite of the performance, we now could apply this pre-trained model to the real-time system. In the general architecture, after the conversion of Spark, data of KDDCup99 format is fed to AI model (stage 6 in figure 1.1). This model predicts that which type of attacks each sample of data belongs to (stage 7 in figure 1.1). The predicted results are sent back to Kafka server. We define a function called `predict(data)` as shown in listing 2.20 to do this work.

```
1  import numpy as np
2  from dl_src.preprocessing import pre_data, get_label_dict
3
4  def predict(model_1, model_2, rf_model, data):
5    results = []
6    in_data = pre_data(data)
7    predictions = model_1.predict(np.array(in_data))
8    predictions = model_2.predict(predictions)
9    predictions = rf_model.predict(predictions)
10
11   label_dict = get_label_dict()
12   for p in predictions:
13     predicted_label = 'normal'
14     for key, value in label_dict.items():
15         if value == p:
16           predicted_label = key
17           break
18     results.append(predicted_label)
19   return results
```

Listing 2.20: Prediction of AI model

Line 2 pre-processes data of KDDCup99 format in order to feed to AI model. This includes the work of transforming category features to numeric values and normalising as presented in the previous semester. However, there is a new problem when a feature in new generated data contains some category values which did not exist before in training data of KDDCup99. We solve this problem by creating new numeric values for them. Data is then fed to model on lines 3-5. We notice that this program is able to process any size of data. `predictions` we obtain from line 9 are still numeric data and they need to be converted to the category values. More specifically, we define a dictionary as shown in listing 2.21. For example, with `predictions = [2, 0, 3]`, lines 12-17 in listing 2.20 convert them to `results = ['r2l', 'normal', 'probe']`.

```
1  def get_label_dict():
2      label_dict = {'normal': 0, 'dos': 1, 'r2l': 2, 'probe': 3, 'u2r':4}
3      return label_dict
```

Listing 2.21: Dictionary of labels

Before sending back these results to Kafka server thanks to a producer, we stick the predicted labels to corresponding samples of data. Up to now, we have data including 41 original features along with their labels. Creating a new producer is similar to the work we present in section 2.2. In short, we finish the discussion of stages 6, 7 and 8 in this section.

2.6 Display results

After the results are returned to Kafka server, we would like to show them in order to warn users (stages 9 and 10). This work is simply done by creating a new consumer with a compatible topic to retrieve results from Kakfa server. The creation of a consumer is presented in section 2.2.

Chapter 3

Results, Discussion and Conclusion

In this chapter, we firstly present the results. Secondly, we discuss some negative points. Finally, we conclude our work.

3.1 Results

All separated components of the whole system are presented in the previous section. We combine them all together to make a real-time system which is able to operate in real network environment. Our video of demo could be found at https://www.youtube.com/watch?v=5G5mZ2JT_jo. We use figure 3.1 captured from this real-time system to illustrate the outcome.

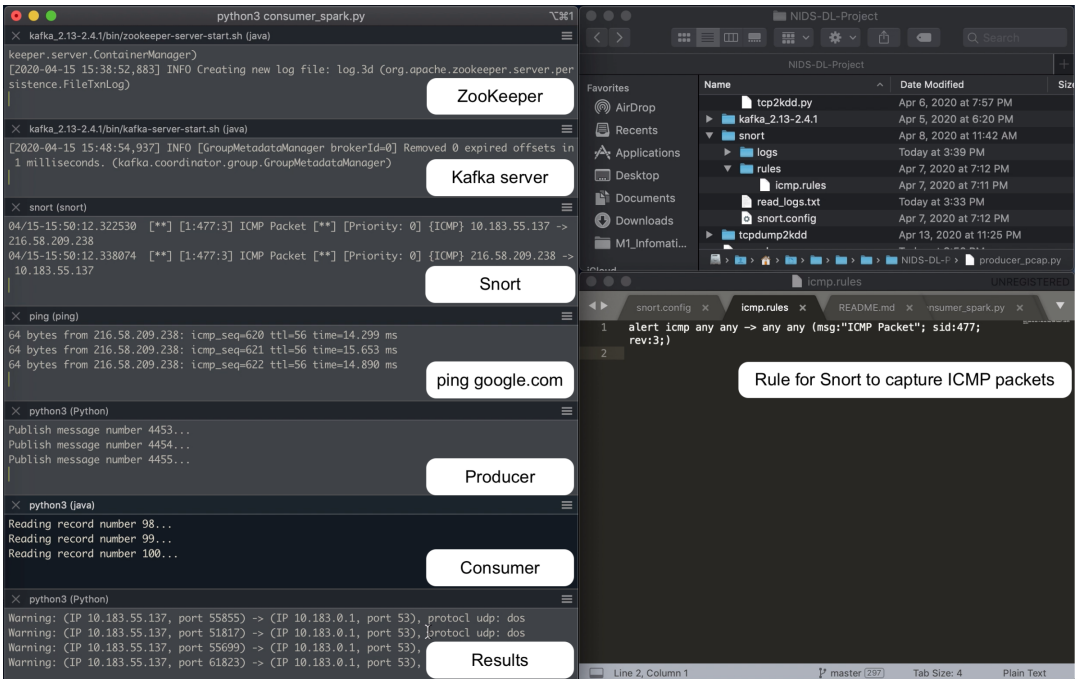


Figure 3.1: Capture of real-time system

In the demo, we define a rule to capture all ICMP packets which go through our computer. We

ping www.google.com to prove that Snort is working well. Producer is activated to send packets captured by Snort to Kafka server which has already been running. A consumer retrieves pcap logs from Kafka server, then processes these logs with Zeek, Spark and AI model. After getting the predictions of AI model, another producer sends the results back to Kafka server. A consumer finally retrieves the results and displays to users.

If we put our attention to the results, they need to be improved in the future. A lot of ICMP packets in the demo are predicted as "dos" attacks. This inaccuracy happens because of the poor performance of the AI model. We will discuss this problem in the next section.

3.2 Discussion

Despite the fact that we achieve the goal of building a real-time network intrusion detection system based on Deep Learning, there are some points that need to be discussed. The first one is the dependant of system's accuracy to AI model. Warnings in the output are trustworthy only if the AI model is good enough. This relates to a lot of things such as the quality of training data, the procedure of processing data or the architecture of model. As far as we know, there are some publications which point out the generation faults of KDDCup99 [1], [4], [6]. Therefore, building an AI model based on KDDCup99 dataset is need to be carefully considered.

The second point we would like to raise in discussion is the redundancy in using Snort and Zeek. Both are network intrusion detection systems. For Snort, it can convert pcap logs to readable logs if combining with *tshark* or *tcpdump*. For Zeek, it is able to monitor network traffic by applying rules as Snort does. If we replace Snort by Zeek at the beginning of the general architecture, Zeek can capture suspect traffic as well as transform traffic to readable data almost at the same time. Hence, it is not necessary to have the stage of transforming pcap to readable data. Therefore, it might be possible to use only one of them.

3.3 Future Work

Our real-time system works well and responses to almost basic requirements posed at the beginning. However, there are still some approaches to develop the current system. The first idea is to continue training the AI model with results of system itself. We depict this idea by stages 11 and 12 shown in figure 3.2. We have data of 41 training features together with labels which are predicted by AI model. These data could be use as a new dataset to train the current model. The more training data we feed to the AI model, the better it is. After a certain amount of time, we train the old model in system with new generated results and replace it by the new one. Nevertheless, the big problem in this idea is the accuracy of labels predicted by AI model. Incorrect training data can lead to a worse model.

The second approach in development is as mentioned in discussion, the combination of Snort and Zeek. Elimination of this redundancy might speed up the computation of the whole procedure. The next idea is to find directions for the limitation of KDDCup99 dataset. For this point, we need to carefully analyse the dataset and survey modern researches.

3.4 Conclusion

In this project, we successfully built a real-time network intrusion detection system based on Deep Learning. We also discussed its limitations in section 3.2 and proposed directions to resolve them in the future in section 3.3. To build the system, we learned and worked with several useful tools such as Snort, Kafka, Zeek, TensorFlow. We come to a conclusion that the AI model plays the most important role in the whole architecture. Increasing the accuracy of AI model is the only

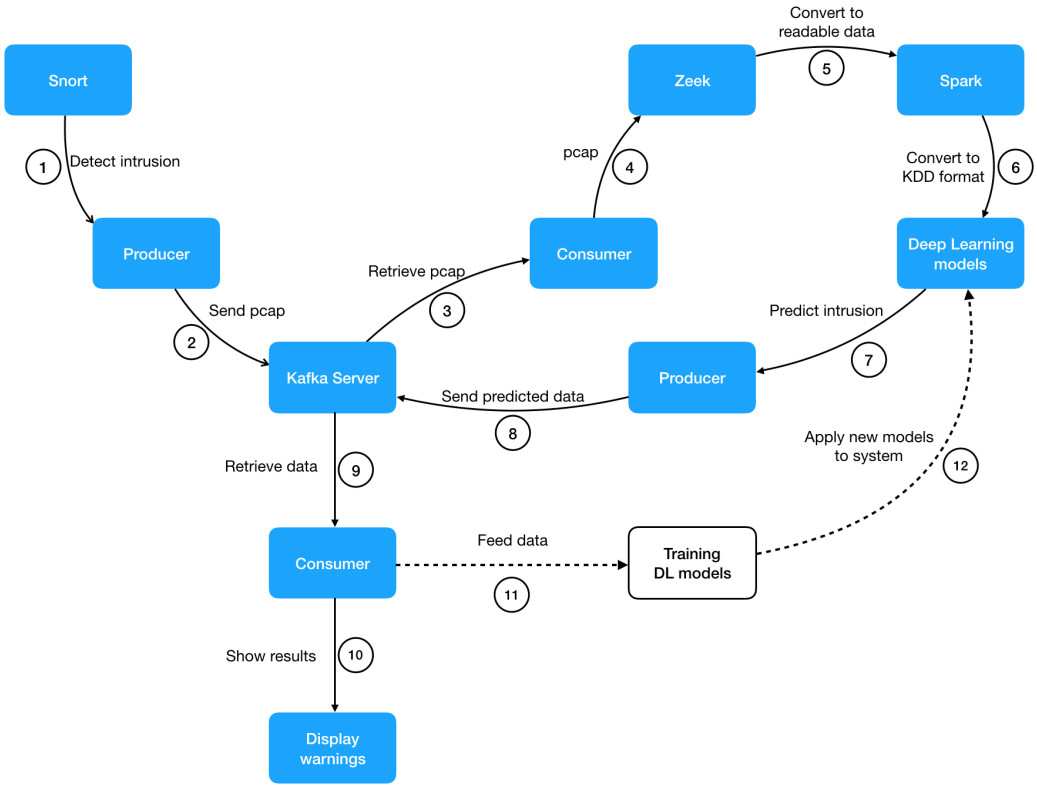


Figure 3.2: A proposed approach of improvement

way to make the system better. Although the performance of system is not excellent as expected, the idea of this project is very applicable to reality.

Bibliography

[1] Al-Tobi, A., Duncan, I.: Kdd 1999 generation faults: a review and analysis. Journal of Cyber Security Technology pp. 1–37 (09 2018). https://doi.org/10.1080/23742917.2018.1518061

[2] N. Shone, T. N. Ngoc, V. D. Phai, Q. Shi: A deep learning approach to network intrusion detection. IEEE Transactions on Emerging Topics in Computational Intelligence **2**(1), 41 – 50 (02 2018)

[3] Panero, P., Vâlsan, L., Brillault, V., Schuszter, I.: Building a large scale intrusion detection system using big data technologies. p. 014 (12 2018). https://doi.org/10.22323/1.327.0014

[4] Pawlak, Z., Polkowski, L., Skowron, A.: Rough Sets and Rough Logic: A KDD Perspective, pp. 583–646. Physica-Verlag HD, Heidelberg (2000). https://doi.org/10.1007/978-3-7908-1840-6_13, https://doi.org/10.1007/978-3-7908-1840-6_13

[5] Shekhar, J., Khurana, A.: Streaming systems and architectures. Spring **41**(1) (2016)

[6] Tavallaee, M., Bagheri, E., Lu, W., Ghorbani, A.A.: A detailed analysis of the kdd cup 99 data set. In: 2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications. pp. 1–6 (2009)

[7] Kdd cup 1999. (Ocotber 2007), available on: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html