ANOMALOUS NETWORK PROTOCOLS WITH MACHINE LEARNING AND ZEEK

Trevor Longmire

A Capstone Project Submitted to the
University of North Carolina Wilmington in Partial Fulfillment
of the Requirements for the Degree of
Master of Science

Department of Computer Science
Department of Information Systems and Operations Management

University of North Carolina Wilmington

May 2023

Approved by

Advisory Committee

Dr. Geoffrey M. Stoker

Dr. Gulustan Dogan

Dr. Jeffrey W. Cummings, Chair

Accepted By

Dean, Graduate School

TABLE OF CONTENTS

Page

Chapter 1: Introduction....................4

Chapter 2: Review and Analysis....................5
 Current Process....................6
Chapter 3: Methodology....................8
 Talking to Zeek....................8
 Broker....................10
 Logging....................11
 Training data....................14
 Normalization....................16
 Autoencoder....................21
 Predictions....................25

Chapter 4: Outline of Completed Project....................32
 Introduction....................32
 Review and Analysis....................32
 Methodology....................32

Chapter 5: Conclusions and Future Work....................34
 Collecting and Normalizing Data....................34
 Lessons Learned....................35

References....................37

Tables
 1 Training Data Protocols....................39

ABSTRACT

Anomalous Network Protocols with Machine Learning and Zeek. Longmire, Trevor, 2023. Capstone Paper, University of North Carolina Wilmington.

The USDA Office of the Chief Information Officer (OCIO) has requested the Cyber Defense Operations Division (CDOD) provide a fully scalable system to monitor network traffic in real time. Continuous monitoring on a large-scale organizational network requires strategy, timely information and 24/7 vigilance. The Infrastructure Security Center (ISC) within CDOD is tasked to provide ongoing solutions to manage risk within the network. ISC uses a comprehensive tool suite for alerting and reacting to a variety of network intrusion attempts. Some of those tools include Zeek intrusion detection systems (IDS), and security event management systems (SIEM). There are many other systems for integration and correlation of data and the licensing costs of these systems are enough to explore the possibilities of having an open-source alternative. Using Zeek to detect network anomalies will improve visibility of the network while driving down costs.

CHAPTER 1. INTRODUCTION

Intrusion detection is a key component of cybersecurity that involves monitoring network traffic and identifying suspicious activity that may indicate a security breach. The primary goal of intrusion detection is to detect and prevent unauthorized access to computer systems and networks with minimal false positives. Many modern-day security operation centers use a variety of techniques to detect security threats, including:

1. Signature-based detection: This approach uses a database of known attack signatures to identify malicious activity. The IDS compares network traffic to the database of signatures to determine if any matches are found.

2. Anomaly-based detection: This approach identifies unusual behavior or patterns in network traffic that may indicate a security breach. This type of IDS uses machine learning algorithms to detect anomalies and determine if they represent a security threat.

This capstone focuses on how to do real-time network classification using Zeek and Python. The goal is to investigate and analyze the traffic being sent from strategic points across a live network and be able to distinguish one protocol from another with machine learning. We will look at methods of gathering bits of raw network packets by breaking them down into a data model that Tensorflow or Scikit learn can make use of. This ultimately will allow an analyst to verify different protocols on a live network with a considerable degree of certainty.

CHAPTER 2: REVIEW AND ANALYSIS

Intrusion Detection Systems (IDS) are a security technology that monitors network traffic and identifies suspicious activity that may indicate a security breach. Zeek IDS is an open-source intrusion detection and network monitoring tool that provides real-time analysis of network traffic. Zeek is a highly flexible open-source tool and can tailor to different users who interact between different systems. It is designed to provide visibility into the network, identify security threats, and provide incident response information. The benefits of using machine learning in intrusion detection include:

1. Anomaly detection: Machine learning algorithms can identify unusual behavior or patterns in network traffic that are indicative of a security breach. This is particularly useful for detecting zero-day attacks, which are attacks that exploit vulnerabilities that are not yet known to security experts.
2. Improved accuracy: Machine learning algorithms can analyze large amounts of data and identify patterns that may be difficult for human analysts to detect. This results in improved accuracy in detecting security threats.
3. Automated response: Machine learning algorithms can be trained to automatically respond to security threats in real-time, reducing the response time and improving the overall efficiency of the IDS.
4. Adaptation: Machine learning algorithms can learn from their mistakes and adapt to new types of attacks, improving their accuracy over time.

In summary, Zeek IDS combined with machine learning algorithms can provide a robust and effective solution for intrusion detection and network security monitoring. The use of machine learning in intrusion detection can help improve accuracy, reduce

response time, and adapt to new types of attacks, making it an essential tool for modern network telemetry. (Bagui et al., 2022)

Current Process

The Infrastructure Security Center (ISC) currently uses Zeek in a limited capacity by aggregating and storing large amounts of log files mandated by data retention policies. The logs are being shipped directly to a SIEM, and it is up to an analyst to manually input a complicated query of an index or database once an event has occurred for further inspection. There are other vendors who provide threat intelligence feeds across their proprietary products, but this can be prohibitively expensive across a network of this scale. Many vendors also claim that their products already have machine learning built into them that detect "events", but many of these solutions are signature based. Some incident response teams like to look at log files after an indicator of compromise for policy violations, but this can be impractical. Of course, there is no right way to handle everything, but having an idea of what exactly is going up and down the network is better than trying to track down lateral movement after the fact. A more sought-after approach would be to predict or catch the event before it is about to happen.

The USDA has a network of over 700,000 endpoints. There are approximately 40TB of logs per day being stored over an array of SAN devices until a proper data lake is architected. (Baldi, 2012, pp. 27-30) The current production environment consists of a cluster of 30 Zeek sensors and 1 manager with 1 backup. All systems are running on a version of Red Hat (RHEL) over a range of 10Gb to 100Gb fiber connections. The development cluster is comprised of dedicated machine learning boxes which all have Python virtual environments activated. Jupyter notebook has been installed for the remote integrated development environment and there is also the ability to SCP files over to a

local Windows machine for debugging. There are currently live packets being fed across a listener as a broker communication channel between the Zeek cluster. The deliverable of this project will be to create a data model to provide a neural network comprised of inputs and outputs for an autoencoder, or otherwise known as an anomaly detector. The intent of this project is to create a more streamlined approach for incident response teams and analysts to gain a better understanding of what's traveling across their network.

CHAPTER 3: METHODOLOGY

Building an auto encoder with Python will allow classification between protocols and can distinguish differences between those protocols. There is a way to create a program that interprets data being fed by Zeek, then output the data in a model that can be further processed. The idea is to take a TCP stream from the Zeek cluster using a broker communication channel and then push the data to tensorflow or scikit learn. This will allow further processing by the autoencoder for anomaly detection, which is usually relegated into the task of unsupervised learning. (Zaccone & Karim, 2018, p.24)

Talking to Zeek

We're going to look at a live feed of data that is cumulatively being sniffed over various interfaces by building a broker communication channel. Broker is the communications library that is built into Zeek. ("Zeek Documentation," 2023) It is how all the workers or "sensors" talk between the manager and the log server. Hooking into broker is one of the most interesting features within Zeek because not only can you automate what Zeek does within its own language, you can also push this data out anywhere and do what you want with it. The device cluster is already running, so there must be a small broker script that fires out every stream that it sees. Python essentially grabs what is being put out by the broker stream, parses the data and prints the output.

```
1  module Tensorflow;
2
3  global query: event(orig_h:addr, orig_p:port, resp_h:addr, resp_p:port, message:dns_msg);
4
5  event zeek_init()
6  {
7  # if(Cluster::local_node_type() == Cluster::MANAGER){
8    Broker::subscribe("tensorflow/content");
9    Broker::listen("127.0.0.1", 9997/tcp);
10   Broker::auto_publish("tensorflow/content", query);
11 # }
12 }
13
14 event dns_message(c: connection, is_orig: bool, msg: dns_msg, len: count)
15 {
16   Broker::publish("tensorflow/content", query, c$id$orig_h, c$id$orig_p, c$id$resp_h, c$id$resp_p, msg);
17 }
18
```

Figure 1. Comms.zeek file

The comms.zeek file allows communication between the Zeek framework and the broker library. The broker library can then enable applications to communicate with Zeek's type-rich data model. In figure 1, the short script is to set up endpoints that represent data senders and receivers. It is essentially an abstracted socket that accommodates communication between the listener and endpoints within the cluster. Once the script was complete, it was run from a Tmux terminal to listen on interface eno1 from the Zeek backup server which is also part of the Zeek cluster.

```
[root@iscipxx0002 bin]# /opt/zeek/bin/zeek -C -i eno1 /opt/ML/THP_ML/comms1.zeek
listening on eno1
```

Figure 2. Zeek listener

The next step is to create a small broker script in Python that communicates with the broker library. Python will grab what is being put out by the broker stream, parse the data and print the output. The data can be accessed in a variety of formats, but Zeek is very resistant on sending binary content anywhere, so the initial output will be sent in ascii and rendered out as hex.

Broker

The TCP connection stream has been established between the Zeek cluster and the broker API, there is a library in Python that communicates between the endpoint that is also called broker. This essentially creates a socket that subscribes to the endpoint. It then can run on an infinite loop by getting a real-time stream of data of the selected arguments from the Zeek data feed. In figure 3 below, these arguments are originating host, originating host port, responding host, responding host port, along with the content or message. As seen in figure 1.

```
import sys
sys.path.insert(0, "/opt/zeek-4.0.5-release/lib64/zeek/python")
import broker
import numpy as np

endpoint = broker.Endpoint()
subscription = endpoint.make_subscriber("tensorflow/content")
status_subscription = endpoint.make_status_subscriber(True)
endpoint.peer("127.0.0.1", 9997)

status = status_subscription.get()

if not (type(status) == broker.Status and status.code() == broker.SC.PeerAdded):
    print("Connection failed")
    sys.exit(1)

print("Connected")

while 1:
    (tag, data) = subscription.get()
    #print(tag,broker.zeek.Event(data).args())
    #print(tag,data)
    (src, sport, dst, dport, content) = broker.zeek.Event(data).args()
    print(content_to_features(content))
```

Figure 3. Zeek broker

Logging

The output initially streamed from broker is rendered out in a ascii-hex representation at first. During testing, the output was rendered through Python's *bytearray* function to show a comparison of the output in figure 4 below. The content variable is used as an argument in the print statement from Figure 3 to show the header in this case. This is going to ultimately be normalized for the auto encoder later.

```
'tcp -> :9998/tcp   :: :: bytearray(b'\x16\x03\x01\x00\x8f\x01\x00\x00\x8b\x03\x03S\xbb!+
\x05\xc24\r\x8c}\xef%\xf6\xe2\xf8\xcc\xf7\xdf\xf7\xe3\xd4.\xd6\xec"\x0bF\x13\x10\xa0\xa2~\x00\x00 \xc0,\xc00\xc0+\xc0/\xc0$\x
c0(\xc0#\xc0\'\x00\x9d\x00\x9c\x00<\xc0.\xc0-\xc0&\xc0%\x00\xff\x02\x01\x00\x00A\x00\x0b\x00\x04\x03\x00\x01\x02\x00\n\x00\x0
8\x00\x06\x00\x17\x00\x18\x00\x19\x00#\x00\x00\x00\r\x00 \x00\x1e\x06\x01\x06\x02\x06\x03\x05\x01\x05\x02\x05\x03\x04\x01\x04
\x02\x04\x03\x03\x01\x03\x02\x03\x03\x02\x01\x02\x02\x02\x03\x00\x0f\x00\x01\x01')
```

Figure 4. Initial output

Transport layer security (TLS) is a widely used cryptographic protocol that provides secure communication over computer networks. Since most data is being encrypted over HTTPS, we will be using this data as a baseline for our training model. This will be used in determining the presence of other protocols such as HTTP, DNS, and SSH. We will be having a closer look at what this data looks like using the Cyber chef tool, which is available on GitHub. Cyber chef is a useful tool to break and inspect various forms of data in multiple formats. In this case, we will break down the hexadecimal byte string to its raw value as seen below in figure 5. From the TLS training data file, the data was imported to the clipboard and converted from hex to raw output. There will be discussion about how we generate the train data later on, but first its important to know what you are seeing here in order to gain insight in how this training data will be processed in order to build the auto encoder.

Figure 5. Raw output from TLS data

This is the actual stream data that was sent to us by Zeek. By examining this data, we can see that there are commonalities in the beginning during the first few bytes, but everything after becomes random from our point of view. It does not matter if its still part of the application header or if it starts to become encrypted because there is still a header we can use. The idea is not to learn how to parse this information out, but to train the neural network to identify that part of the header so you can later run other data against the model, such as data from an HTTP header or DNS record. Note the clear text in the output such as HTTP or drop box. There will be commonalities like this in the model, but after training thousands of lines of data, the autoencoder will still be able to make a distinction between TLS and HTTP. Let’s compare figure 5 to an HTTP file showing the differences in figure 6. The same thing can be done for DNS and SSH data, but it will look slightly different.

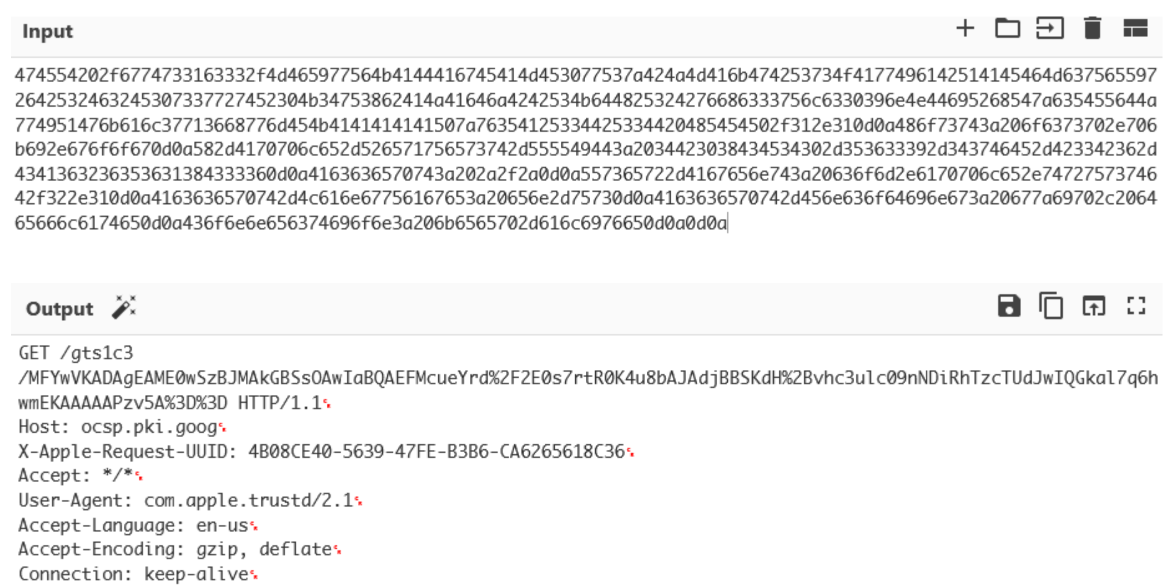


Figure 6. Raw output from HTTP data

Notice the 474554 in the input string reads GET. This is not a big surprise because most queries begin with GET followed by a space. Most characters are also ascii characters as opposed to the binary characters that we had in our TLS data.

Training data

Zeek can also be configured to log strings from multiple protocols using the Python broker library. These logs are saved in a training data folder for further processing. The collected data is then normalized and prepared for utilization in the machine learning model. To accomplish this, a modification was made to another Python script, enabling the original ASCII-hex data to be written to specific files based on their respective ports. The objective is to organize the data into a directory where it can be easily accessed for normalization.

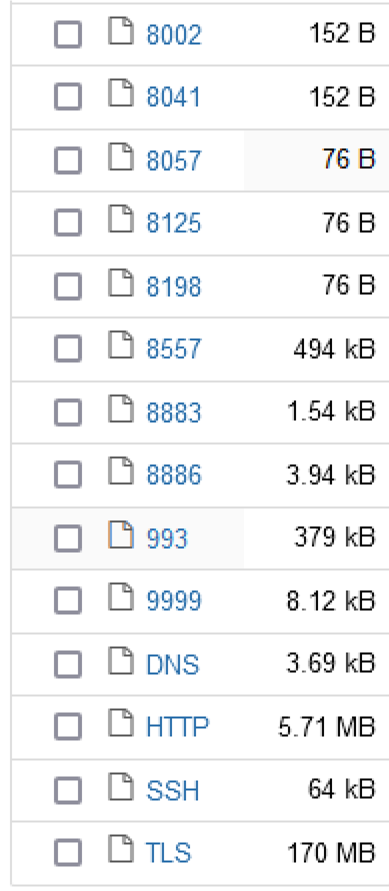
During the experiment, a "get-data" script was executed for a short period while the listener was active. This script was concise, resembling the structure depicted in Figure 3, but with alterations to the while loop. As a result, a substantial amount of training data, approximately 170 megabytes, was obtained, consisting of various protocols. However, for the purpose of this research, only a few selected protocols will be tested and used.

This approach facilitates the collection and organization of network traffic data for training purposes, offering a meaningful dataset of 170 megabytes. By focusing on specific protocols, the research aims to evaluate the effectiveness of the model in accurately identifying and classifying different types of network traffic.

Overall, this process enables the acquisition, normalization, and organization of training data from multiple protocols, laying the groundwork for subsequent analysis and model development. Figure 7 shows how the data was streamed directly into a directory with separate files for each protocol. Some of the protocols were changed for purposes of development.

```python
while 1:
    (tag, data) = subcription.get()
    (src, sport, dst, dport, data) = broker.zeek.Event(data).args()
    with open (f'training_data/{dport.number()}', 'a') as f:
        f.write(f'{data}\n')
```

Figure 7. Get data with HTTP file example



1 474554202f6261673f69783d3220485454502f312e310d0a486f73743a20696e69742d7030316d642e6170706c652e636f6d0d0
65702d616c6976650d0a782d6c6f6767696e673a20747275650d0a4163636570743a202a2f2a0d0a557365722d4167656e743a2
634f532c31312e352e322c32304739352c4d61636d696e69392c315d0d0a782d696e7465726e616c3a20747275650d0a416363
6e2d75730d0a4163636570742d456e636f64696e673a20677a69702c206465666c6174650d0a0d0a
2 474554202f6261673f69783d3220485454502f312e310d0a486f73743a20696e69742d7030316d642e6170706c652e636f6d0d0
63636570742d4c616e67756167653a20656e2d75730d0a436f6e6e656374696f6e3a206b6565702d616c6976650d0a41636365
69702c206465666c6174650d0a557365722d4167656e743a207365727665722d626167205b6950686f6e65204f532c31342e372
335d0d0a0d0a
3 474554202f677473163332f4d465977564b4144416745414d453077537a424a4d416b474253734f41774961425141455464d637
52304b34753862414a41646a4242534b6448253242766863333756c6330396e4e44695268547a635455644a774951476b616c377
35412533442533442048545450 2f312e310d0a486f73743a206f6373702e706b692e676f6f670d0a582d4170706c652d526571717
434534302d353633392d343746452d423342362d434136323635363138433336 0d0a4163636570743a202a2f2a0d0a557365722
6c652e7472757374642f322e310d0a4163636570742d4c616e67756167653a20656e2d75730d0a4163636570742d456e636f646
6174650d0a436f6e6e656374696f6e3a206b6565702d616c6976650d0a0d0a
4 474554202f67747372312f4d464d7755614144416745414d456f77534442474d416b474253734f417749614251414546444435277
646d7752755a25324642514242546b7279736d63526f72534365464c314a6d4c4f2532467769524e785067494e41674f3855316
25334420485454502f312e310d0a486f73743a206f6373702e706b692e676f6f670d0a582d4170706c652d526571756573742d5
373831342d344443422d383431422d36333930454336454539453 90d0a4163636570743a202a2f2a0d0a557365722d4167656e7
757374642f322e310d0a4163636570742d4c616e67756167653a20656e2d75730d0a4163636570742d456e636f64696e673a206
436f6e6e656374696f6e3a206b6565702d616c6976650d0a0d0a

Normalization

Data normalization is a crucial step in preparing data for machine learning models. In the context of this research, normalizing the data involves converting ASCII-hex lines into a binary representation. The primary goal of normalization is to eliminate any confusion or bias that may arise during the training process.

By converting the data into bits, the complexity and variability of the original ASCII-hex lines are reduced, creating a standardized format that can be efficiently processed by machine learning algorithms. This standardization helps ensure that the model does not favor certain patterns or values due to the specific representation of the data.

One example highlighting the importance of normalization is the narrowing down of packet headers for processing. In this case, failing to normalize the data could lead to false negatives, where certain patterns or information in the packet headers are not accurately captured or considered by the model. Normalization helps mitigate such issues by providing a consistent representation that captures relevant information without introducing biases or inconsistencies.

Furthermore, it is crucial to ensure that the machine learning model is capable of normalizing the data and storing it in a suitable format for later use. The scaling or normalization process involving packet headers requires careful consideration, as these headers can vary in length. By scaling the data to a 0 to 1 relationship, the model can effectively handle the varying lengths and interpret the bits within the headers consistently.

```
MAX_LENGTH = 32

def content_to_features(data, number_of_bytes = MAX_LENGTH):
    x = data.ljust(number_of_bytes * 2, "0")
    x = x[:(number_of_bytes*2)]
    byte_values =[]
    for i in range(0, len(x), 2):
        byte_values.append(chr(int(x[i:i+2],16)))
    bits = []
    for byte in byte_values:
        bits.append([(ord(byte) & (1<<i))>>i for i in [7,6,5,4,3,2,1,0]])
    return list(np.array(bits).astype(np.uint8).flat)
```

Figure 8. Conversion function

There now needs to be a conversion function which is shown in Figure 8 that converts the string of hexadecimal ascii characters to a list of bits. The function takes an input string and an optional number of bytes, then converts each pair of characters of the string into a single byte. The 32-byte variable was chosen arbitrarily for the neural network to have a constant number of features for normalization. Finally, the rest of the function converts each byte into a list of 8 bits and returns them as a flat list. The function will then be called by an infinite loop in figure 3 that continually retrieves data from the subscription and parses the data using the global Zeek event function.

```
for filename in files:
with open(f'{TRAINING_DATA}/{files[0]}', 'r') as f:
    lines = f.read().splitlines()             # read as an array with split
    for line in lines:
        print(f'{filename},{",".join(map(str,content_to_features(line)))}')
```

```
while 1:
    (tag, data) = subscription.get()
    # Grab whats being put out by the broker stream, parse it out and print
    (orig_h, orig_p, resp_h, resp_p, content) = broker.zeek.Event(data).args()
    direction = "->"
    o = f'{orig_h}:{orig_p}'
    r = f'{resp_h}:{resp_p}'
    if(orig_h == False):
        direction = "<-"
    string = content_to_features(content)
    print(f'{o:>25} {direction} {r:25} :: :: {string}')
```

Figure 9. Looping through live subscription service

The output from the content argument is passed through the conversion function. The obfuscated ip addresses with the protocol are shown only for testing purposes and the final output will be shown below in figure 10.

```
/tcp -> /tcp  :: :: [0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0,
0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 1,
1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 0, 0, 0,
0, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 1, 1,
1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0,
1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1]
```

Figure 10. Stream array of bits

There is now a live feed of data consisting of 256 bits converted from the original 32 bytes of each TCP header. This should show how the data is being parsed while we prepare it for our final output in figure 11.

```
[0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0,
1, 1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,
0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1,
1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1,
0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 1, 1,
0, 0, 1, 0, 1]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0, 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, 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, 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, 1, 0, 1, 0, 1, 1,
1, 1, 1, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0,
1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1, 1, 1, 1, 0,
1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1,
1, 0, 0, 1, 1]
```

Figure 11. Final normalized data

The next step is to build a training dataset that can distinguish different protocols between the bits. There are different ways to access this data for processing, but for the purposes of testing and development the data will be stored in memory.

```python
with open('.\\training_data\\TLS', 'r') as f:
    raw_data = [content_to_features(line) for line in f.readlines()]
```

Figure 12. Accessing normalized data

The training data begins by pulling the entire TLS file and running each line through the conversion function content_to_features. NumPy is a fundamental library in the field of data science due to its numerous advantages. It provides efficient and optimized numerical operations on multi-dimensional arrays, enabling faster computations and memory utilization for data preprocessing. The raw data variable is passed to x_data in figure 13.

```python
x_data = np.array(raw_data).astype(np.uint8)
x_data.shape # output
```

```
(197400, 256)
```

Figure 13. X_data training set

```
x_data[0]
```

```
array([0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0,
       0, 1, 0, 0, 0, 0, 0, 0, 1, 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, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1,
       1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1,
       0, 1, 0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1,
       1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 1,
       1, 0, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 0,
       0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1,
       1, 1, 0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 1,
       1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 0, 1,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 1, 0, 0,
       1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0], dtype=uint8)
```

Figure 14. Numpy array of bits

The data type uint8 are unsigned 8 bit integers (Zaccone & Karim, 2018, p. 50) converted to a numpy array of bits from raw_data which is 197,400 lines of TLS data. Each index is 256 bits which was sliced to 190,000 lines to build the x_train variable. The x_test is everything after the 190,000 which would be 7,400 lines.

```
x_train = x_data[:190000]  # train up to the first 190,000 rows
x_test = x_data[190000:]   # splits these 2 apart
```

Figure 15. Test training set

Autoencoder

```
import tensorflow as tf
from tensorflow.python.keras import Sequential
from tensorflow.python.keras.layers import Dense

#from tensorflow.keras import Sequential, layers
#from tensorflow.keras.layers import Dense
import numpy as np
```

Autoencoders are a type of neural network used in machine learning for various tasks, such as dimensionality reduction, feature extraction, and anomaly detection. In this context, the standard API is recommended over the function API for building the autoencoder model. The model is initialized as Sequential, allowing for a straightforward layer-by-layer construction. A dense layer is added, connecting all neurons in the layer to the neurons in the next layer. In this case, a dense layer with 64 neurons is chosen, and the Rectified Linear Unit (ReLU) activation function is utilized. Another layer is added to account for the input shape.

The model architecture consists of four dense layers with 64, 32, 16, and 64 neurons, respectively. AE networks can have multiple layers, where the input layer and the output need to have the same number of neurons. (Zaccone & Karim, 2018, p.24) The ReLU activation function is applied to each dense layer. The final output layer is another dense layer with the appropriate number of neurons.

```
# Max length of 32 bytes * 8 bits = total bits in each sample.
input_dimensions = MAX_LENGTH * 8

model = Sequential()  # feed forward model
model.add(Dense(input_dimensions, input_shape=(input_dimensions,)))  # , to make a tuple
model.add(Dense(input_dimensions // 2, activation='relu'))  # // int division
model.add(Dense(input_dimensions // 4, activation='relu'))  # relu = rectified linear unit
model.add(Dense(input_dimensions // 8, activation='relu'))
model.add(Dense(input_dimensions // 4, activation='relu'))
model.add(Dense(input_dimensions // 2, activation='relu'))
model.add(Dense(input_dimensions, activation='relu'))
```

Figure 16. The autoencoder

```
# summary of the model
model.summary()

model.compile(optimizer='adam', loss='mae', metrics=['accuracy'])
model.fit(x_train, x_train, batch_size=128, epochs=10, verbose=1)

#evalute the model
#test_loss, test_acc = model.evaluate(x_test, x_test)
#print('Test accuracy:', test_acc)
```

Figure 17. Summarize and run model

In Figure 17. this first line of code, model.summary( ) prints a summary of the model's architecture, displaying information about the layers, their output shapes, and the total number of parameters in the model. The next function is model.compile( ). Here, the model is compiled and configured for training. The compile function sets the optimizer, loss function, and evaluation metrics for the model. In this case, the optimizer is set to Adam, which is a popular optimization algorithm. The loss function is set to 'mae', which stands for Mean Absolute Error, and it is used to measure the difference between the predicted output and the true output. The metrics parameter is set to ['accuracy'], which means that the model's accuracy will be monitored during training. The model.fit( ) function initiates the training process. This trains the model using the provided training data x_train. In this case, the input and output data are the same (x_train is passed twice), which suggests that the model is being trained for an autoencoder task or a reconstruction task. The batch_size parameter specifies the number of samples to be used in each update of the model's weights. The epochs parameter indicates the number of times the model will iterate over the entire training dataset. In this code snippet, the model will train for 10 epochs. Finally, the verbose parameter is set to 1, which means that progress updates will be printed during training.

During each epoch, the model will process the training data, make predictions, calculate the loss based on the chosen loss function, and update its internal parameters using the optimizer. The goal is for the model to gradually improve its performance by minimizing the loss. After completing the specified number of epochs, the training process will end, and the model can be evaluated or used for making predictions on new, unseen data.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                Output Shape              Param #
=================================================================
dense (Dense)               (None, 64)                4160

_________________________________________________________________
dense_1 (Dense)             (None, 32)                2080

_________________________________________________________________
dense_2 (Dense)             (None, 16)                528

_________________________________________________________________
dense_3 (Dense)             (None, 8)                 136

_________________________________________________________________
dense_4 (Dense)             (None, 16)                144

_________________________________________________________________
dense_5 (Dense)             (None, 32)                544

_________________________________________________________________
dense_6 (Dense)             (None, 64)                2112

=================================================================
Total params: 9,704
Trainable params: 9,704
Non-trainable params: 0
```

Output of figure 17.

```
Total params: 9,704
Trainable params: 9,704
Non-trainable params: 0
_________________________________________________________________
Epoch 1/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0477 - accuracy: 0.1459
Epoch 2/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0435 - accuracy: 0.1423
Epoch 3/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0348 - accuracy: 0.1034
Epoch 4/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0338 - accuracy: 0.1161
Epoch 5/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0337 - accuracy: 0.1151
Epoch 6/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0337 - accuracy: 0.1230
Epoch 7/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0337 - accuracy: 0.1240
Epoch 8/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0336 - accuracy: 0.1101
Epoch 9/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0336 - accuracy: 0.1120
Epoch 10/10
1485/1485 [==============================] - 2s 1ms/step - loss: 0.0336 - accuracy: 0.1090

<tensorflow.python.keras.callbacks.History at 0x2c90d3039a0>
```

Output of figure 17 cont.

Predictions

Testing or showing how well the model works is going to require visualization of the bits in the training data folder to compare how different they are to the predictions dataset in our model. So far there is x_test data to do predictions on with our model. This data is simply the TLS data that was used in figure 15.

```
predictions=model.predict(x_test)
#predictions.dtype
```

```
predictions
```

```
array([[0.        , 0.        , 0.        , ..., 0.18916246, 0.        ,
         0.9790814 ],
        [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
         0.9802406 ],
        [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
         0.98114306],
        ...,
        [0.        , 0.        , 0.        , ..., 0.26919538, 0.        ,
         0.        ],
        [0.        , 0.        , 0.        , ..., 0.53771794, 0.        ,
         0.        ],
        [0.        , 0.        , 0.        , ..., 0.        , 0.        ,
         0.989033  ]], dtype=float32)
```

Figure 18. Predictions from x_test

Remember back in figure 7 where there was a TLS folder with approximately 170mb of data in it. This was the 197,000 lines of TLS data that we collected from our listener. The training model was built as a base line with this data of the first 190,000 lines and the remainder of those lines were sliced to a x_test variable. What predictions is essentially showing is the mean absolute error of the x_test data to the x_train data, which you can see in figure 19 that there is not much of a difference when comparing that data. What there needs to be is comparison between other data types, such as HTTP or SSH data.

```
import matplotlib.pyplot as plt
plt.plot(tf.losses.mae(x_test, predictions))
plt.show()
```

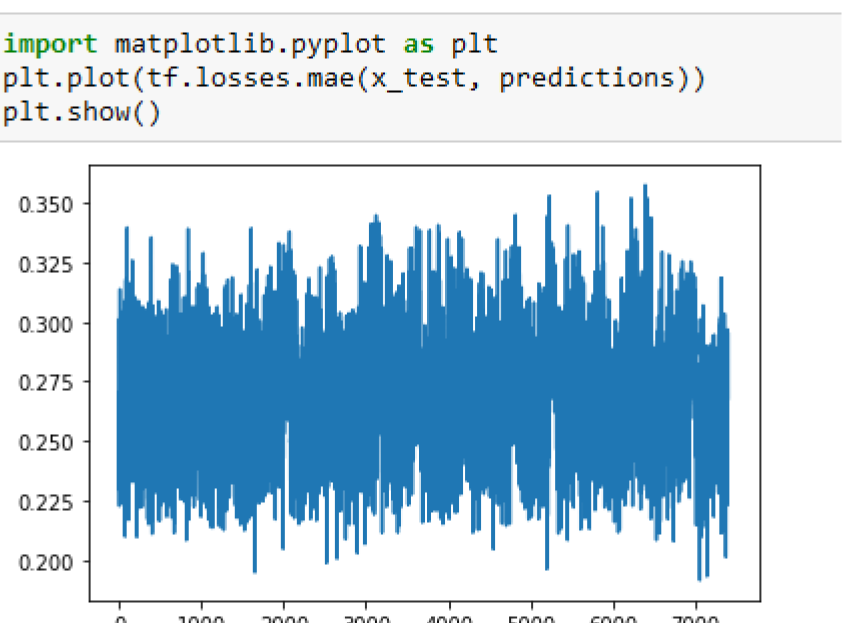


Figure 19. Predictions against x_test

Opening the HTTP file from our training folder, there are 15,261 lines of data with 256 columns representing each bit. For each line, the data was normalized after running through our conversion function. When we compare this data to TLS data there should be similarities as well as differences. The TLS test set is 7400 lines of data, which is good enough for running this kind of observation.

```
with open('.\\training_data\\HTTP', 'r') as f:
    raw_data = [content_to_features(line) for line in f.readlines()]
```

```
http_data = np.array(raw_data).astype(np.uint8)
http_data.shape
```

```
(15261, 256)
```

```
predictions.shape
```

```
(7400, 256)
```

```
plt.plot(tf.losses.mae(x_test, predictions))    #use for http data

plt.plot(tf.losses.mae(http_data[:7400], model.predict(http_data[:7400])))

plt.show()
```

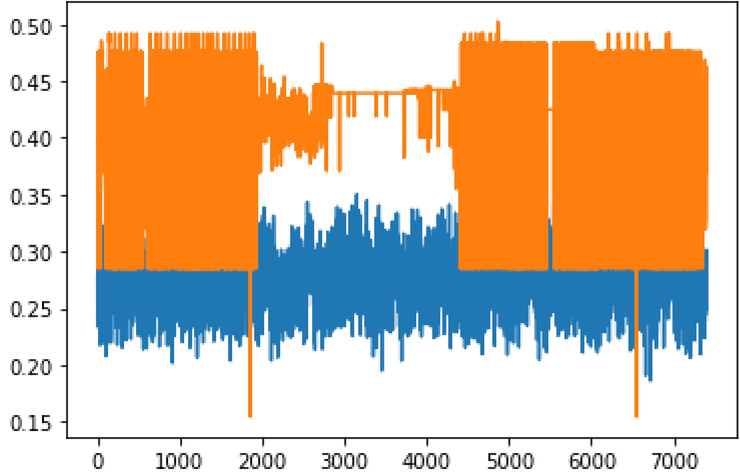


Figure 20. Plotting HTTP data to TLS data

Note that there is some overlap between the orange HTTP data and the blue TLS data. One of the objectives of this research is compare the training data to other data that may come in from a feed to find linear separation, or in other words, to find a clear distinction between various data types. Note in the snippet below from figure 8 in the conversion function where the max_length is equal to 32 bytes. The variable was set to 16 and the test was run again to see if there were different results. The results indicated there were still false negatives, but each subsequent iteration with the number of bytes in each header was narrowed down to yield better outcomes.

```
MAX_LENGTH = 32

def content_to_features(data, number_of_bytes = MAX_LENGTH):
    x = data.ljust(number_of_bytes * 2, "0")
    x = x[:(number_of_bytes*2)]
```

Figure 8. Conversion function

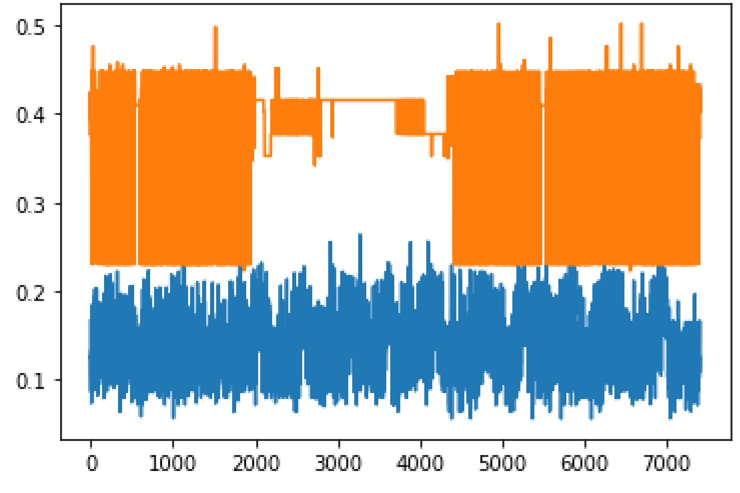


Figure 21. TLS and HTTP at 16 bytes

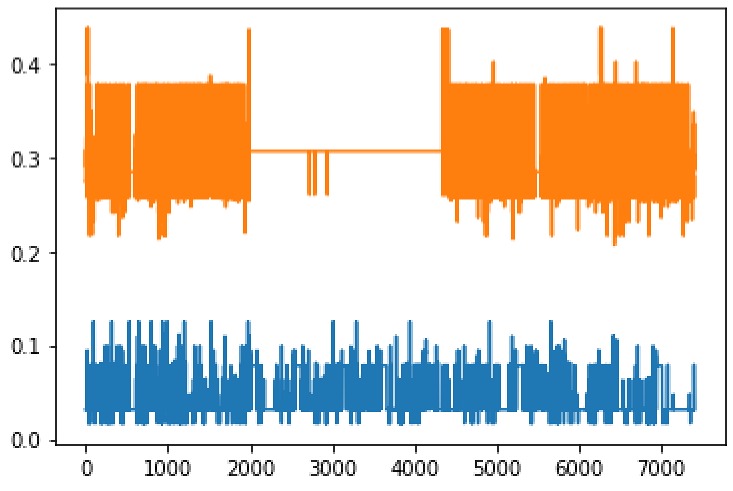


Figure 22. TLS and HTTP at 8 bytes

On figure 21 between 2500 and 4500 there are instances where the TLS data crosses into HTTP territory and where HTTP crosses into TLS. Looking back to cyberchef again, we can confirm what this will look like.

Output

SYNETXSOHSTXNULSOHNULSOHÜETXETX^GÏ»ả¹ŸûVTDC3±Í§.e•Ð•• ••
vy7Æ•q•Ñú STX÷yNULVTESC|«•0BELSOH\7r`%•SYNÑæ•4kÈÁÏ.•• EM 'NUL6ZZDC3SOHDC3STXDC3ETXÀ,À+Ì©À0À/Ì¨À$À#À
À À(À'ÀDC4ÀDC3NUL•NUL•NUL=NUL<NUL5NUL/À BS ÀDC2NUL
SOHNULSOH}zzNULNULNULNUL FS NULSUBNULNULETBdownload-a.akamaihd.netNULETBNULNULÿSOHNULSOHNULNUL
NUL FF NUL
BS http/1.1
°•0ÞDC4\ÁÝj>•'ESCöÙ·=µ2SUBeòÆP·¤>L SO p•
°°ETXEOTETXETXSTXETXSOHúúNULSOHNULNULNAKNULµNULNULNULNULNULNULNULNUL

Figure 23. TLS header at 32 bytes

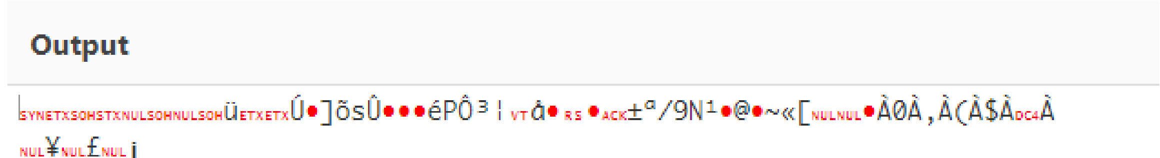


Figure 24. TLS header at 16 bytes

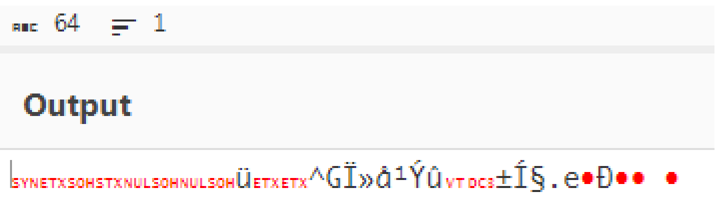


Figure 25. TLS header at 8 bytes

64 1

Output

GET /images/social-icons/website

Figure 26. HTTP header at 8 bytes

Looking at this header in figure 23 there are instances of plain text and encrypted data in the 32 byte header. This explains the overlap in our training model when HTTP data was ran against the original TLS training data using 32 bytes for the conversion function. The header was trimmed down to 16 bytes but since our dataset was over 190,000 lines, there were still instances in the header where there was clear text visible.

Narrowing down the header to 8 bytes was good enough to eliminate any instances of where clear text was present in the TLS header. Finally in figure 26, this is essentially what the data looks like when it is being run against the output in figure 25.

The other training data was smaller for obvious reasons but were still captured for the purposes of this research. There were 49 lines of DNS traffic collected, and 1392 lines of SSH traffic. Pulling the data and running against the TLS training dataset took only a few lines of code.

```
with open('.\\training_data\\DNS', 'r') as f:
    raw_data = [content_to_features(line) for line in f.readlines()]
```

```
dns_data = np.array(raw_data).astype(np.uint8)
dns_data.shape
```

```
(49, 64)
```

```
predictions.shape
```

```
(7400, 64)
```

Figure 27. DNS dataset

Here in figure 27 the DNS data was normalized using the conversion function and converted to a numpy array of 49 rows and 64 columns. The header was left at 8 bytes instead of 32 since there are better results when narrowing the header for training. The output in figure 28 shows clear linear separation between DNS and TLS data. The same exact process was done with SSH data which is also provided in figure 29.

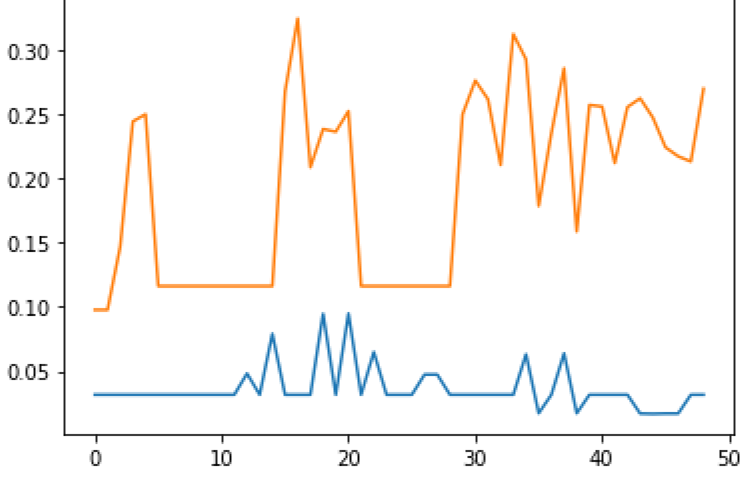


Figure 28. DNS and TLS

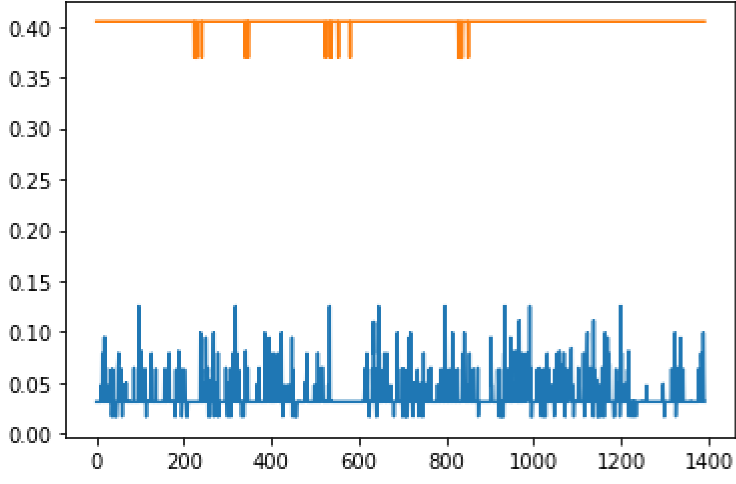


Figure 29. SSH and TLS

CHAPTER 4: OUTLINE OF COMPLETED PROJECT

Introduction:

This is an overview of intrusion detection in cybersecurity, emphasizing its goal of detecting and preventing unauthorized access with minimal false positives. The chapter introduces signature-based and anomaly-based detection techniques commonly used in security operation centers. It also outlines the capstone project's focus on real-time network classification using Zeek and Python, aiming to analyze network traffic, distinguish protocols with machine learning, and enable confident protocol verification on live networks.

Review and Analysis:

In this section, the review and analysis provide an overview of intrusion detection systems in general. Zeek IDS is a popular choice being an open-source tool for real-time network traffic and is widely known as an industry standard. This chapter then dives in and highlights the benefits of incorporating machine learning for intrusion detection. Anomaly detection provides a means for improved accuracy, automated response, and adaptation to new attack types. It emphasizes the synergy between Zeek IDS and machine learning algorithms, showcasing their potential to enhance intrusion detection, reduce response time, and adapt to evolving security threats in modern network security strategies.

Methodology:

This research focuses on building an autoencoder with Python for anomaly detection while leveraging Zeek. The chapter explains the process of interpreting data and feeding it into a model for further processing. It explores the utilization of a broker communication channel to extract TCP streams from the Zeek cluster, which are then

pushed to TensorFlow or scikit-learn for processing by the autoencoder. The chapter covers topics such as communication with Zeek through the broker, logging data, training data normalization, and the use of a neural network or autoencoder to make predictions for future network data. Multiple protocols, including TLS, HTTP, SSH, and DNS are tested in this chapter to evaluate the effectiveness of the autoencoder for anomaly detection.

CHAPTER 5: CONCLUSIONS AND FUTURE WORK

Intrusion detection plays a critical role in ensuring cybersecurity by monitoring network traffic and identifying potentially malicious activity indicative of a security breach. The main objective of intrusion detection is to detect and prevent unauthorized access to computer systems and networks while minimizing false positives. Contemporary security operation centers employ various methods for threat detection, including signature-based detection, which involves comparing network traffic against a database of known attack signatures. Anomaly-based detection, on the other hand, utilizes machine learning algorithms to identify abnormal behavior or patterns in network traffic that may indicate a security threat. This research focuses on real-time network classification using Zeek and Python. The study aims to investigate and analyze traffic from strategic network points, leveraging machine learning techniques to differentiate between different protocols. By extracting and transforming raw network packet data into a format compatible with Tensorflow or Scikit-learn, analysts can verify protocols on live networks with a high level of confidence.

Collecting and Normalizing Data

The availability of training data sets is crucial for the effectiveness of machine learning models in the specific network environment. In this research, Zeek, along with the Python broker library, enables the logging and organization of strings from multiple protocols into a training data folder. By collecting and normalizing this data, the research aims to evaluate and optimize the model's ability to accurately identify and classify network traffic in the specific network environment. The acquisition of tailored training data sets enhances the model's performance, leading to improved network traffic analysis and security in the targeted network environment.

In summary, normalizing the data by converting ASCII-hex lines into a binary representation is essential for machine learning models. It helps eliminate confusion, bias, and inconsistencies, enabling more accurate and reliable training. Additionally, careful consideration should be given to normalizing data involving packet headers, ensuring proper scaling to a 0 to 1 relationship to accommodate variations in length and maintain data integrity.

Dense layers play a crucial role in the construction of autoencoder models used in machine learning. They connect all neurons in a layer to the neurons in the next layer, enabling comprehensive information flow and representation learning. In the provided context, a dense layer with 64 neurons and the Rectified Linear Unit (Relu) activation function is chosen, facilitating effective data transformation. The inclusion of multiple dense layers with varying neuron counts allows for the extraction of increasingly complex features. The careful selection and configuration of dense layers are pivotal in achieving optimal performance and accuracy in autoencoder models. (Zafar, Tzanidou, Burton, Patel & Araujo, 2018, p.32) Autoencoders are valuable in machine learning as they enable dimensionality reduction, anomaly detection, and representation learning, thereby aiding in tasks like data preprocessing and unsupervised feature extraction.

Lessons Learned

The development of a production-ready anomaly detection system presents both setbacks and potential areas for improvement. One setback encountered was licensing issues with Red Hat Enterprise Linux (RHEL), which may have hindered the deployment and utilization of certain components of the system. Additionally, changes to the Security Technical Implementation Guide (STIG) for Linux operating systems posed challenges in maintaining the system's compatibility and adherence to security standards.

Moving forward, a crucial aspect to address is the categorization of each protocol within the data model. This categorization is necessary for the creation of a Tensor Board callback, which would enable the machine learning model to continuously learn and update its data model based on the feedback received. Achieving this capability will enhance the system's adaptability and accuracy in detecting anomalies.

Once the categorization and Tensor Board integration are implemented, the focus can shift towards optimizing the system for speed and scalability across an enterprise. This involves fine-tuning the algorithms, improving the efficiency of data processing, and ensuring the system can handle the increasing volume of network traffic in a robust and timely manner.

As with many software projects, the initial phase of getting the system to work serves as a foundation. The subsequent stages will involve collaboration with a team of software developers using agile methodologies. This collaborative effort facilitates ongoing changes, modifications, and refinements to the system, ensuring its continuous improvement and alignment with evolving requirements and industry best practices.

REFERENCES

Waleed, A., Jamali, A. F., & Masood, A. (2022). Which open-source IDS? Snort, Suricata or Zeek. Computer Networks, 213, 109116.

Bhuyan, M. H., Bhattacharyya, D. K., & Kalita, J. K. (2013). Network anomaly detection: methods, systems and tools. Ieee communications surveys & tutorials, 16(1), 303-336.

Baldi, P. (2012, June). Autoencoders, unsupervised learning, and deep architectures. In Proceedings of ICML workshop on unsupervised and transfer learning (pp. 27-30). JMLR Workshop and Conference Proceedings.

Pang, B., Nijkamp, E., & Wu, Y. N. (2020). Deep Learning With TensorFlow: A Review. Journal of Educational and Behavioral Statistics, 45(2), 227–248.

Hao, J., & Ho, T. K. (2019). Machine Learning Made Easy: A Review of Scikit-learn Package in Python Programming Language. Journal of Educational and Behavioral Statistics, 44(3), 348–361.

Bagui, S., Mink, D., Bagui, S., Ghosh, T., McElroy, T., Paredes, E., ... & Plenkers, R. (2022). Detecting reconnaissance and discovery tactics from the MITRE ATT&CK framework in Zeek Conn Logs using Spark’s machine learning in the big data framework. Sensors, 22(20), 7999.

Zaccone, G., & Karim, M. R. (2018). Deep Learning with TensorFlow: Explore neural networks and build intelligent systems with Python. Packt Publishing Ltd.

Zafar, I., Tzanidou, G., Burton, R., Patel, N., & Araujo, L. (2018). Hands-on convolutional neural networks with TensorFlow: Solve computer vision problems with modeling in TensorFlow and Python. Packt Publishing Ltd.

Zeek Documentation. (2023, April 1). Zeek Documentation - Book of Zeek (Git/Master). Retrieved from https://docs.zeek.org/en/master/

Zeek Documentation. (n.d.). Retrieved April 5, 2023, from Zeek Documentation website: https://docs.zeek.org/projects/broker/en/lts/overview.html

TABLE 1

		Training Data Protocols	
Protocol	**Port**	**Bits**	**Bytes from Zeek**
HTTP	80	256	32
TLS	443	256	32
SSH	22	256	32
DNS	53	256	32
Changes in header length determining if the protocols are linearly separable			
HTTP to TLS		64	8
SSH to TLS		64	8
DNS to TLS		64	8

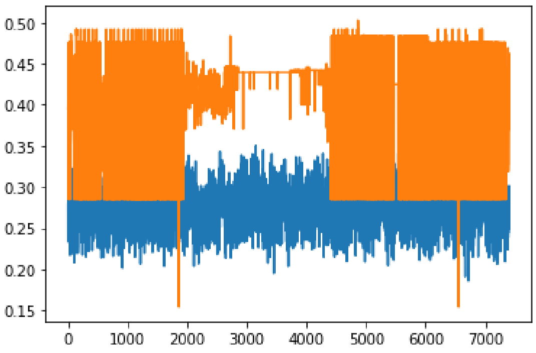


Figure 20. Plotting HTTP data to TLS data

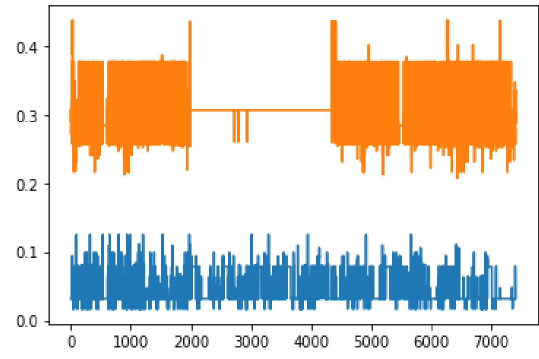


Figure 22. TLS and HTTP at 8 bytes

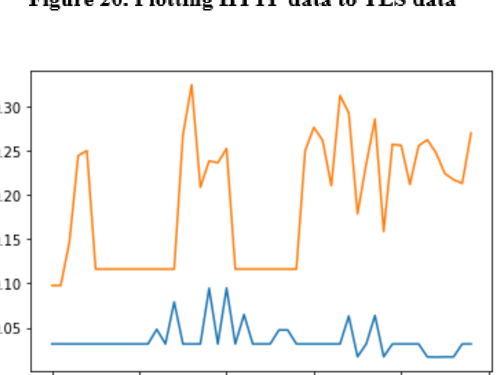


Figure 28. DNS and TLS

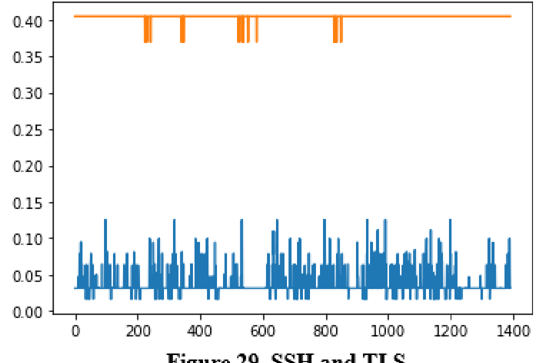


Figure 29. SSH and TLS