NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

INFERRING NETWORKING EVENTS FROM TRANSPORT LAYER SECURITY–ENCRYPTED TRAFFIC

by

Cardavian J. Lowery

June 2021

Thesis Advisor: Geoffrey G. Xie
Second Reader: John D. Fulp

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*

1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2021	**3. REPORT TYPE AND DATES COVERED** Master's thesis

4. TITLE AND SUBTITLE INFERRING NETWORKING EVENTS FROM TRANSPORT LAYER SECURITY–ENCRYPTED TRAFFIC	**5. FUNDING NUMBERS**
6. AUTHOR(S) Cardavian J. Lowery	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000	**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A	**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release. Distribution is unlimited.	**12b. DISTRIBUTION CODE** A

13. ABSTRACT (maximum 200 words)

Security protocols are one of the most secure ways to ensure an outsider threat does not gain access to information sent across networks. Current security protocol standards typically encrypt packet payloads against such intrusions. But with data encryption comes new challenges to monitor communication on a network. In Software Defined Networks (SDN), Transport Layer Security (TLS) is commonly used to encrypt OpenFlow messages exchanged between a controller and each switch under its control. TLS results in lack of data visibility to network monitors and this, in-turn, can prevent timely detection of and response to various network events. In this thesis, we develop solutions to classify encrypted OpenFlow traffic into OpenFlow message types. It examines the effectiveness of two traffic classification techniques using a dataset generated from a simulated SDN, and shows that the techniques can achieve an accuracy up to 95%. The most successful features used to classify encrypted OpenFlow messages are explained along with a methodology of collecting data, labeling data, identifying features, and the training of models to achieve high accuracy of classification.

14. SUBJECT TERMS Transport Layer Security, TLS, Software Defined Networks, SDN, inferencing, machine learning, classification	**15. NUMBER OF PAGES** 65
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	**18. SECURITY CLASSIFICATION OF THIS PAGE**	**19. SECURITY CLASSIFICATION OF ABSTRACT**	**20. LIMITATION OF ABSTRACT**
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release. Distribution is unlimited.

INFERRING NETWORKING EVENTS FROM TRANSPORT LAYER SECURITY–ENCRYPTED TRAFFIC

Cardavian J. Lowery
Lieutenant, United States Navy
BS, The Citadel, 2014

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2021

Approved by: Geoffrey G. Xie
Advisor

John D. Fulp
Second Reader

Gurminder Singh
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Security protocols are one of the most secure ways to ensure an outsider threat does not gain access to information sent across networks. Current security protocol standards typically encrypt packet payloads against such intrusions. But with data encryption comes new challenges to monitor communication on a network. In Software Defined Networks (SDN), Transport Layer Security (TLS) is commonly used to encrypt OpenFlow messages exchanged between a controller and each switch under its control. TLS results in lack of data visibility to network monitors and this, in-turn, can prevent timely detection of and response to various network events. In this thesis, we develop solutions to classify encrypted OpenFlow traffic into OpenFlow message types. It examines the effectiveness of two traffic classification techniques using a dataset generated from a simulated SDN, and shows that the techniques can achieve an accuracy up to 95%. The most successful features used to classify encrypted OpenFlow messages are explained along with a methodology of collecting data, labeling data, identifying features, and the training of models to achieve high accuracy of classification.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

AI	Artificial Intelligence
DHCP	Dynamic Host Configuration Protocol
IPSec	Internet Protocol Security
NPS	Naval Postgraduate School
SDN	Software Defined Networking
SSH	Secure Shell Protocol
SSL	Secure Socket Layer
SVM	Support Vector Machine
TCP	Transmission Control Protocol
TLS	Transport Layer Security
Tor	The Onion Router
UDP	User Datagram Protocol
VM	Virtual Machine
WPA	Wi-Fi Protected Access

THIS PAGE INTENTIONALLY LEFT BLANK

Acknowledgments

First and foremost, I would like to give all praises to God, without whom, none of this is possible.

I would like to thank Dr. Geoffrey Xie and Professor John Fulp for all their support and advice. Your guidance and counsel have been invaluable during this journey.

I would also like to thank the United States Navy for giving me and my family the opportunity to spend two fantastic years at NPS in California.

Finally, I would like to express my gratitude to my wife and daughter for their support and great patience with my long hours in front of my computer.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1:
Introduction

Network traffic is digitally-formatted information that traverses one of more computer based networks. This traffic is broken into discrete packets that represent the smallest units of conveyance for carrying data and/or protocol-related control information. Any packet or sequence of packet or sequence thereof that has an effect on the network is referred to as a network event. These events can range from requesting an IP address from the dynamic host configuration protocol (DHCP) server, to requesting a cached web page from a web server. These events are the ongoing operations of any given organization. The technique of inferring network events from live or captured network traffic is commonly referred to as network traffic analysis or classification. Accurate network traffic classification is advantageous in network management and administration as it can provide timely indications and warning regarding potential malicious activities such as network intrusions or data exfiltration. Classification of unencrypted network traffic has been studied with promising results [1]. However there has been limited research of classifying encrypted packets for traffic analysis purposes. Encryption of network traffic provides an essential security control for information transiting any of the myriad of networks that constitute the global Internet. It provides for the information security objectives of both confidentiality and integrity thus helping to prevent unauthorized disclosure, modification, or impersonation of information. This thesis considers a specific subset of encrypted packet flows, Transport Layer Security (TLS) protected traffic as it applies to Software Defined Networks (SDNs). An SDN is a networking standard which provides the flexibility of programming network services as per the need and demand of user applications. SDNs utilize TLS for control communications between a logically centralized controller and each switch handling user traffic. The ability to infer network events from TLS encrypted data will allow monitoring of SDN control traffic in transit and potentially give new insight into classifying other types of encrypted traffic.

1.1 Inferring Network Events

Network traffic analysis refers to a class of techniques for collecting and examining computer communication traffic for the purpose of detecting and responding to security threats. Because adversaries are continuously modifying their attack methods to avoid detection, it is important for network operators to deploy robust traffic analysis systems.

The study conducted by Menuka et al. [2] details the work conducted to infer network events from unencrypted data. It outlines the process of data preparation (identifying what features would be used for the data), data clustering (labeling the data), classification of the data, and a performance evaluation.

The study conducted utilized a Kaggle data set [3]. The data set was real world, diverse, and was created by collecting network traffic from a university in Popayan, Colombia. Only a subset of the data set was used for the study. Utilizing unsupervised learning for labeling, the data was clustered based on the possible correlation of the data. Labels are created and used to train classification models. Five classification models (Support Vector Machine, Linear and Radial Based Function, Decision Tree, Random Forest, and Kth Nearest Neighbor) are implemented and return various levels of performance. The study conducted a field test integrating machine learning with software defined networks. The study demonstrates how classification algorithms return positive results within the SDN environment.

Traffic analysis becomes difficult when we introduce the paradigm of encryption into the realm of analysis. Due to various malicious activities conducted over the Internet, it becomes prudent to encrypt data traversing the composite networks. The Onion Routing (Tor) Project, Secure Socket Shell (SSH), Wi-Fi Protected Access (WPA), and Internet Protocol Security Protocol Encapsulating Security Payload (IPSec ESP); among other initiatives, provide these capabilities. Having the capability of identifying and classifying encrypted network data and then correlating that data to a specific network event provides a means to more effectively manage and monitor a networks carrying such traffic. This capability is even more significant for larger and more geographically dispersed networks.

What makes the classification of encrypted network traffic stand out from the classification of unencrypted traffic is the increased difficulty of identifying features for classification algorithms to key on. For unencrypted data, models can utilize key information (source IP

address, destination IP address, protocol port number) to create models for classification of future data. With encrypted packets, little to none of this information is visible due to the obfuscation wrought by the encryption. Therefore, other key features of the packets must be identified and employed to accurately create a usable classification model.

1.2 Problem Statement

This thesis investigates whether it is feasible to classify encrypted TLS traffic within an SDN into categories that can serve as indicators of network events. SDNs are well suited to the growing field of cloud services and data centers owning to their adaptability, scalability, security, and logical control; and are thus more capable of enhanced centralized control due to the separation of the control-plane and data-planes [4]. It is the observation of the control plane and its encrypted data that ultimately facilitated collection of the data necessary to derive the results presented in this study. The control plane, by default is unencrypted. By implementing the TLS-related control plane packets, we can by inference and deduction analyze the encrypted control plane traffic. This encryption method is achieved by a TLS proxy.

"A TLS proxy is used in secure connections to allow for additional networking services while protecting against denial-of-service attacks. TLS (Transport Layer Security) provides encryption and authenticity of communication over the Internet. It started out for secure online e-commerce transactions and has quickly become the defacto security protocol. TLS proxies are becoming more prominent than older SSL (Secure Socket Layer) proxies when it comes to handling incoming TLS connections" [5]. Figure 1.1 depicts a simple diagram of a SDN with a pair of TLS proxy connections between the controller and four switches.

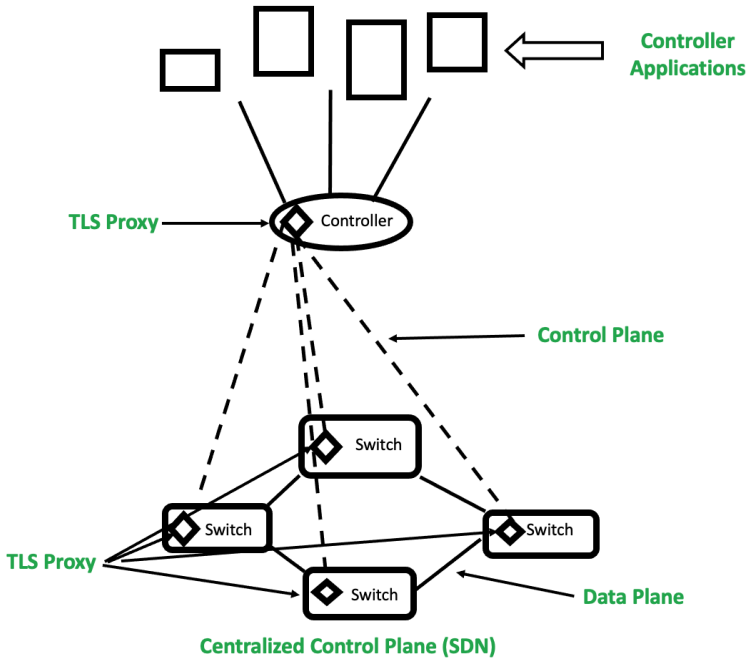


Figure 1.1. SDN with TLS Proxies for Encryption

1.3 Research Questions

The hypothesis being evaluated in this thesis is: *Identifying and classifying encrypted TLS data from SDNs to infer network events will provide an effective method of identifying potential network vulnerabilities and attacks.*

The primary research questions pursued are:

- Does selecting a specific classification algorithm (e.g. k-nearest neighbor, random forests, or decision trees) impact inference accuracy?
- Does the size and topology of the network affect the accuracy and speed of the classification?
- Will the classification solution be deployable in Navy platforms?

1.4 Organization of Thesis

The thesis is organized in five chapters. Chapter 1 provides an introduction to the area of research, the problem addressed, and the scope and purpose of the thesis. In Chapter 2, the reader gets an overview of the most important technology areas related to the thesis. Classification algorithms are described in general terms and TLS, SDN-based TLS technology in described in more detail. Chapter 3 outlines the methodology of the experimental, to include diagrams of the setup and running of the network. The results of the experiment are provided and described in Chapter 4. The thesis concludes with Chapter 5 and an appendix, which also offers considerations for follow-on research as well as source code for the interpreter.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 2:
Background

This chapter provides an overview of the three major technology areas pertinent to this thesis research. First, an introduction to Software Defined Networking (SDN) technology is presented. This section includes detailed information about the Transport Layer Security (TLS) protocol, as well as the Mininet network emulation software used in this work. The second section describes machine learning concepts and classification algorithms relevant to this study. Finally, this chapter ends with a discussion of related work in the area of classifying network activities from traffic analysis.

2.1 Software Defined Networking

A Software Defined Networking entails organizing and managing networks in a way that among other things, separates the configuration channel from the data flow channel. The configuration channel is the *control plane*. The control plane allows for a centralized authority to configure multiple data flow channels that comprise the entire network. The work is handled and managed by the controller, hence the name control plane. The controller handles configuration management of all the SDN compliant devices. The data flow is the *data plane*. This data plane represents the traversal of user related data packets across the network. The switch devices within the network can rely on the control plane to make forwarding decisions or make those decisions on their own. Figure 2.1 shows the architecture of an SDN. There is only one control plane for the management and configuration of the hardware devices. This makes the administration, scalability, and organization of an SDN efficient.

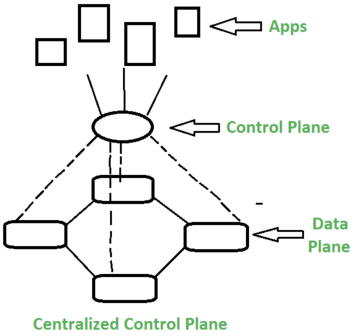


Figure 2.1. Architecture of an SDN

Traditional networks require the use of fixed function network devices. This means that traditional networks are hardware based, as compared to an SDNs which are software based. Traditional networks also have distributed control. This means every hardware device, e.g. router or switch is independently controlled. Figure 2.2 depicts the architecture of a traditional network. From this illustration, it can be seen that an individual control plane is provided by each individual device. This, then places configuration management requirements at each device; which makes the management of these traditional networks cumbersome and tedious.

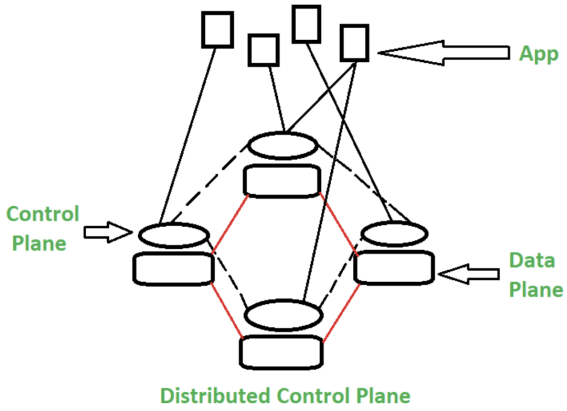


Figure 2.2. Architecture of a Traditional Network

2.1.1 TLS

Transport Layer Security is a prevalent cryptographic protocol. It is used to conceal information that traverses networks. From conducting financial transactions to logging into

Facebook, TLS is one of the backbones of internet security. TLS provides privacy and integrity of data during the communications of system. There are three versions of TLS: TLS 1.1, TLS 1.2, and TLS 1.3. TLS 1.1 [6] is now deprecated. TLS 1.2 [7] is deprecated as well but did introduce better hash algorithms. TLS 1.3 [8] is the current version in use today and adds more security mechanisms such as one round trip handshake and session hashes.

TLS provides a mechanism for networked entities to perform authentication; that is proving that their provided identities are valid/truthful. With public (and private) key encryption, website authentication can be facilitated. TLS also protects data from man-in-the-middle attacks by leveraging symmetric encryption algorithms. Finally message authentication codes are used to facilitate the detection of unauthorized changes to data during traversal across a network.

Within an SDN encryption is provided by TLS. The TLS protocol will create a secure connection from client to server to allow transference of encrypted data. The TLS 1.2 connection process steps are outlined in Figure 2.3.

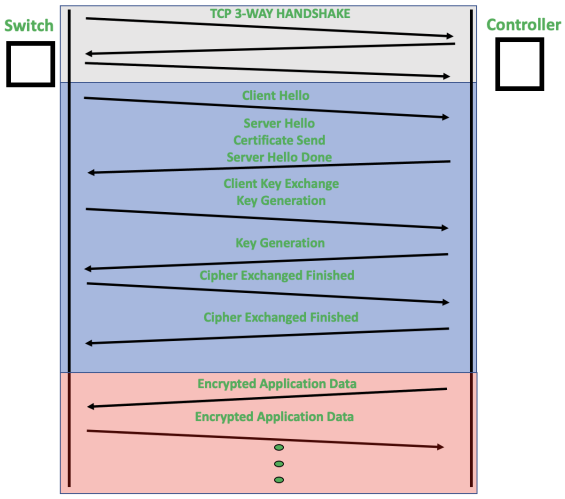


Figure 2.3. TLS 1.2 Connection Establishment

A more in-depth timing diagram of TLS 1.3 message transferal is outlined in Chapter 3.

2.1.2 Mininet

The software program utilized for this study is Mininet. Mininet presents a software based virtual environment in which networks can be created for research or real-life use. All hosts, controllers, switches and protocols execute just as they do in traditional physical networks. Configuration is centralized through a command terminal. OpenFlow is a supported protocol that provides communication between the control and data plane.

The emulation of networks provides a robust and flexible solution to visualize networks when the resources for a physical network are not available. The virtual networks can be created, utilized, and tore down in minutes. This emulation can be achieved using one physical machine (laptop or desktop). Mininet allows you to transmit packets through virtual switches that provide realistic operating metrics, such as with link speed and processing delay. Figure 2.4 shows a graphical depiction of a basic network in Mininet.

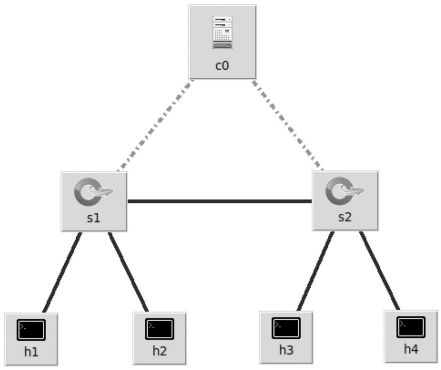


Figure 2.4. Graphical depiction of an SDN in Mininet

2.2 Machine Learning

Machine learning is a subset of artificial intelligence. It is a system in which learning occurs by the observation of data and inputs. The method of learning is achieved by remembering iterations of experiences in order to make the be prediction of a given scenario.

Machine learning is the process that many companies utilize for services they provide. The recommendation system from services like Netflix, YouTube, and Spotify all use machine learning in attempts to deliver the best recommended content for a given user to stream or watch. In this example each service is collecting data about the user. This could include, but is not limited to, what the user is watching, clicking, or what you are reacting to. All of this data is used to make an educated guess on what the user might want to see next.

2.2.1 Supervised and Unsupervised Learning

Machine learning algorithms are generally performed in two ways: supervised and unsupervised. Gathering data, learning the correlations among it and using labels to predict events is supervised learning. The process starts with the analysis of a known training data set. The algorithm yields a model to make predictions about the output. The results can be compared to the ground truth data used to determine correlations and determine the accuracy of the algorithm. A quick example of supervised learning can be seen in the practice of predicting house prices. The key features to be collected would be square footage, number of rooms, and various other amenities, while also noting the prices of similar houses currently or recently on the market. The prediction of new house prices is based on using the composite of available information to deduce the price of a new house (square footage, number of rooms, location, and various other amenities) generated from the data.

Conversely, unsupervised learning algorithms apply to training when the data used is neither classified nor labeled. Since the data is unlabeled, the algorithms can learn different facts about the data and potentially provide new, heretofore unknown or unspecified insights for further analysis. An example of this would be identifying an image that contains cats and dogs. Because the machine has no knowledge of features for these animals, it is not possible to classify the animals. But it can categorize them according to patterns. This method would be used to categorize the image according to the animals' similarities, patterns, and differences.

2.2.2 Classification Algorithms

Since some models can perform better than others perform poorly, it is more advantageous to train and test multiple classification models to determine which is more fitting for the goals of the type of learning being pursued. While there are numerous classification algorithms, Archit Verma outlines several of the most well-known methods for classification that were deemed good candidates for the purpose of this research:

> The decision tree learning algorithm is a top-down, recursive divide-and-conquer, greedy algorithm. A decision tree is a tree like structure in which internal nodes are labeled with attributes and outgoing edges denote outcome of test condition on that attribute while leaf nodes denote classes. Nodes with attributes divide the training dataset into two or more subsets based on the meaning of each related attribute.
>
> The Naive Bayes' classification algorithm is based on Bayes' theorem of posterior probability. This algorithm works by predicting probability that a given data instance belongs to a particular class. Data instance is represented by a vector $X = (x1, x2, x3, ..., xn)$ where $x1, x2, ..., xn$ are the values of the n attributes $A1, A2, A3, ..., An$ in the dataset.
>
> The Support Vector Machine (SVM) algorithm is a supervised learning algorithm that uses labeled data to train the model. The SVM model will calculate decision boundaries between labeled data also known as hyper planes. Points near these hyper-planes are called extreme points. The algorithm will optimize these decision boundaries by setting up margins that separate hyper-planes.
>
> Random forests, also known as random decision forests, are an ensemble learning method for classification, that operate by constructing a number of decision trees at training time and outputting the class that is the majority of the classes outputted from individual trees. Random decision forests correct for decision tree's problem of over fitting to their training data sets. The individual decision trees are created by random selection of tuples from training datasets and random selection of attributes at each split.
>
> K-nearest neighbor (KNN) predicts the class label of an unknown data instance by choosing the majority of the classes labels of K-nearest data instances based

on the distance between the instances as measured by a distance measure formula. KNN is a type of instance-based learning, or lazy learning, where and all computations are done only when the test tuple is presented for classification [9].

2.3 Related Work

Given this thesis entails classification of encrypted data, here are a few studies conducted in this area of research:

The classification described by Wright et al. [10] examined the effectiveness of classification of commonly used protocols that are encrypted. The study displays classifying various encrypted network protocols in two ways. First, it shows the results of classifying network data without reconstructing TCP (i.e. multiple packet) sessions; followed by the results wherein TCP session packets were reconstructed. Comparisons of this data showed that classification accuracy increases when allowing reconstructions of multiple packet sessions owning to such allowing for the development of better heuristics for classification.

Another study administered by Rasteh et al. [11] focused on using machine learning to classify encrypted network traffic while implementing a data classification algorithm. The paper reviews the process of dataset development, data processing, data labeling, and creating a classification model utilizing Spiking Neural Networks (SNNs). The encryption methods used were VPN and Tor. The traffic types classified were web browsing, chat, file transfer, video, and VoIP data. The study demonstrated the ability to classify encrypted data with the SNN classifier to an overall accuracy of 95%.

Another study conducted by M. Sjoholmsierchio [4] demonstrates the ability to add encryption variations to an existing open-source protocol. In his research, M. Sjoholmsierchio creates protocol dialects for the OpenFlow protocol. These dialects standalone from the TLS encryption that can be used by SDNs. The results demonstrated that adding the dialects had no significant impact on latency and overhead.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3:
Methodology

This chapter outlines the methodology of the thesis. It covers the general approach of the research conducted. It describes OpenFlow messaging and its order flow. It provides a brief Mininet walk through. The chapter then covers the encrypted and unencrypted data that will be collected. Next is data labeling followed by feature identification. The chapter concludes by reviewing classification algorithms and which specific classifiers will be utilized.

3.1 General Approach

To answer the research questions, the thesis took a four-step approach:

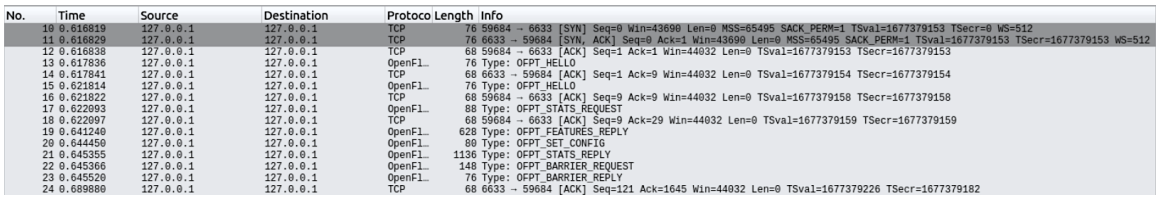
1. The creation of two independent virtual networks. One network implemented no encryption methods, while the other implements TLS 1.3 encryption methods. Both networks are given the same commands to generate network traffic.
2. The pairing of each message from the encrypted network to the unencrypted network to develop heuristics to help identify encrypted messages.
3. The labeling of encrypted messages and identifying features so as to utilize them during sample runs of encrypted traces.
4. The creation of classification algorithms to determine the efficacy the resulting classification.

By both describing the design for a complete solution and prototyping and testing the key components, the thesis addresses the primary research questions. Through the development of the prototype, the key concepts of the solution are tested and some of the challenges of developing the proposed system for TLS encryption are identified and addressed. Furthermore, the prototype is necessary to perform functional evaluation of the proposed solution and also for tests of performance and scalability.

3.2 OpenFlow Events

Before explaining the Mininet configurations, we must observe the OpenFlow packets transmitted across the network. These unencrypted network packets will be the data utilized to create the ground truth for packet analysis.

In the SDN environment, standard OpenFlow protocol communication is used between the controller and the switch. Due to the control and data plane being separated, the controller can focus on overall network information and the switch can focus on data forwarding. Figure 3.1 depicts a Wireshark packet capture during an OpenFlow connection establishment.



No.	Time	Source	Destination	Protoco	Length	Info
10	0.616819	127.0.0.1	127.0.0.1	TCP	76	59684 → 6633 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=1677379153 TSecr=0 WS=512
11	0.616829	127.0.0.1	127.0.0.1	TCP	76	6633 → 59684 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=1677379153 TSecr=1677379153 WS=512
12	0.616838	127.0.0.1	127.0.0.1	TCP	68	59684 → 6633 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=1677379153 TSecr=1677379153
13	0.617836	127.0.0.1	127.0.0.1	OpenFl…	76	Type: OFPT_HELLO
14	0.617841	127.0.0.1	127.0.0.1	TCP	68	6633 → 59684 [ACK] Seq=1 Ack=9 Win=44032 Len=0 TSval=1677379154 TSecr=1677379154
15	0.621814	127.0.0.1	127.0.0.1	OpenFl…	76	Type: OFPT_HELLO
16	0.621822	127.0.0.1	127.0.0.1	TCP	68	59684 → 6633 [ACK] Seq=9 Ack=9 Win=44032 Len=0 TSval=1677379158 TSecr=1677379158
17	0.622093	127.0.0.1	127.0.0.1	OpenFl…	88	Type: OFPT_STATS_REQUEST
18	0.622097	127.0.0.1	127.0.0.1	TCP	68	59684 → 6633 [ACK] Seq=9 Ack=29 Win=44032 Len=0 TSval=1677379159 TSecr=1677379159
19	0.641240	127.0.0.1	127.0.0.1	OpenFl…	628	Type: OFPT_FEATURES_REPLY
20	0.644450	127.0.0.1	127.0.0.1	OpenFl…	80	Type: OFPT_SET_CONFIG
21	0.645355	127.0.0.1	127.0.0.1	OpenFl…	1136	Type: OFPT_STATS_REPLY
22	0.645366	127.0.0.1	127.0.0.1	OpenFl…	148	Type: OFPT_BARRIER_REQUEST
23	0.645520	127.0.0.1	127.0.0.1	OpenFl…	76	Type: OFPT_BARRIER_REPLY
24	0.689880	127.0.0.1	127.0.0.1	TCP	68	6633 → 59684 [ACK] Seq=121 Ack=1645 Win=44032 Len=0 TSval=1677379226 TSecr=1677379182

Figure 3.1. OpenFlow Connection Establishment

The first three packets are the Transmission Control Protocol (TCP) three-way handshake. The connection is initiated by the switch. Once the handshake is established, the switch and controller send *OFPT_HELLO* packets to one another. They each send these messages to identify the highest OpenFlow protocol version it can support. Once the messages have been acknowledged the session has been established.

After session establishment, the controller sends an *OFPT_STATS_REQUEST*, asking for information on which ports are available. The switch responds with an *OFPT_FEATURES_REPLY* message, noting the ports available, their speeds, and supported actions. The switch also sends an *OFPT_STATS_REPLY* that provides detailed information about the switch such as serial number, software description, and hardware description. Next, the controller sends the *OFPT_SET_CONFIG* message. The message includes the flags and maximum bytes allowed to be sent to the controller. The controller then sends an *OFPT_FLOW_MOD* message to modify the state of the switch.

The controller then sends the *OFPT_BARRIER_REQUEST* message. This message is utilized by the controller to determine whether the switch has completed all previous operations. All previously sent messages from the controller must be completed after this message is sent. The switch then responds with an *OFPT_BARRIER_REPLY* to the

controller.

Next, the *OFPT_PACKET_IN* is sent by the switch to the controller. There are three situations when this occurs: no matching flow entry, action request sent to the controller, and invalid Time to Live (TTL) field. The switch also has the ability of sending an *OFPT_PORT_STATUS* to indicate the status change of one of its ports. Finally, in order to check for the liveliness of the connection *OFPT_ECHO_REQUEST* and *OFPT_ECHO_REPLY* are sent from the switch to the controller and vice versa [12]. Figure 3.2 depicts the message timing diagram for OpenFlow.

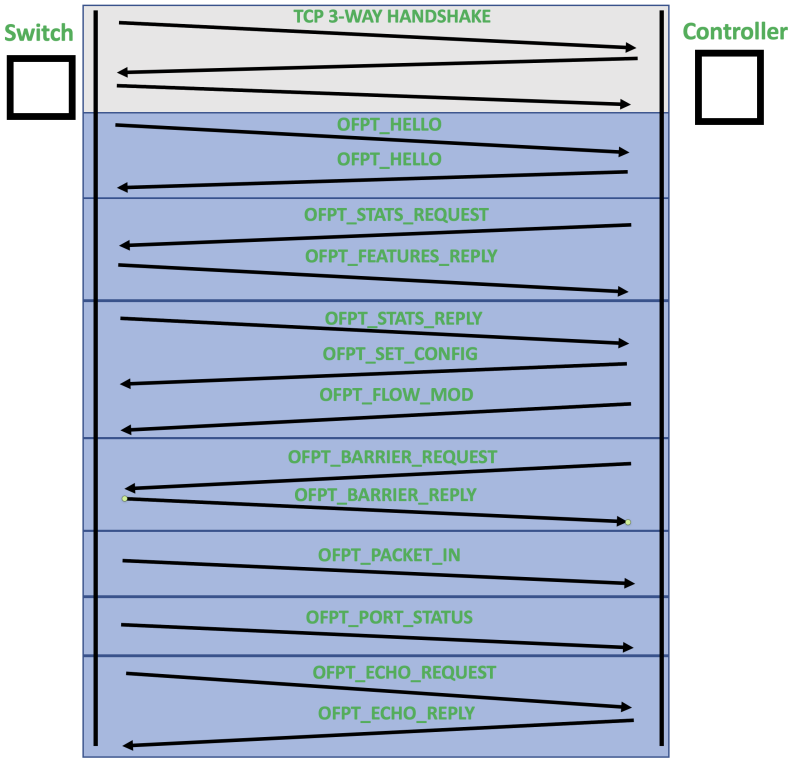


Figure 3.2. OpenFlow Message Diagram

3.3 Mininet Implementation

Having obtained a brief understanding of OpenFlow messages, we can now discuss the implementation of the Mininet to conduct this study. As mentioned previously, the tool Mininet was chosen because of its flexible ability to construct and modify networks quickly. The version of OpenFlow utilized is OpenFlow 1.4. The Ubuntu version used is 18.04.3.

The baseline VMware snapshot is provided by [4]. His study examines protocol dialects within SDNs. This study utilized the already established TLS version 1.3 encrypted network to gather data for classification. It also takes advantage of the scripts created to efficiently rerun network scenarios. Numerous networks were created with various amounts of hosts ranging from 2 to 50. The more hosts within the network, the more data to be generated across the control plane of the SDN.

3.4 Mininet Walkthrough

To run the experiment, two scripts were created. The first script was for the unencrypted network and the second was for the encrypted network. Both scripts executed the exact same code. The only difference is the latter script established a TLS connection between the controller and the switch. The following is the walk through and explanation of executing the unencrypted script.

Opening a terminal and entering:

```
$ sudo ./run_notls test
```

The sudo command allows Mininet to run as root. The rest of the command is the execution of the script file and specifying the output file named *test*. The script takes several actions when it is run. Firstly, it cleans up old files created by previous executions of the code. It then resets Mininet to the baseline network configuration. Once the cleanup is done, the controller is started. Then, Wireshark is started and listens to port 6633. Port 6633 is the port the controller will be listening to the switch on. The switch will listen to the controller on a randomly assigned ephemeral port. Finally the switch is started and connected to the controller. The switch will start with a predefined network topology configuration. The network will run for some predetermined time and shutdown. The controller, Wireshark, and the switch will shut down. What is left is a packet capture detailing all the packets sent across the control plane. Figure 3.3 displays the script execution.

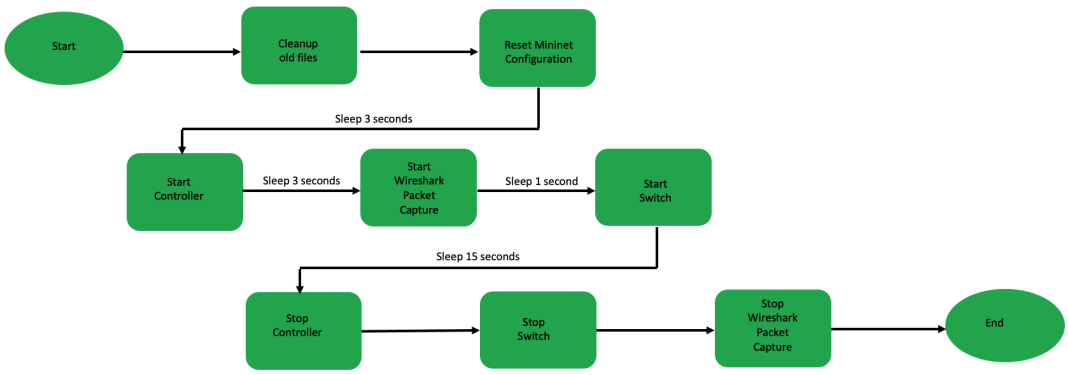


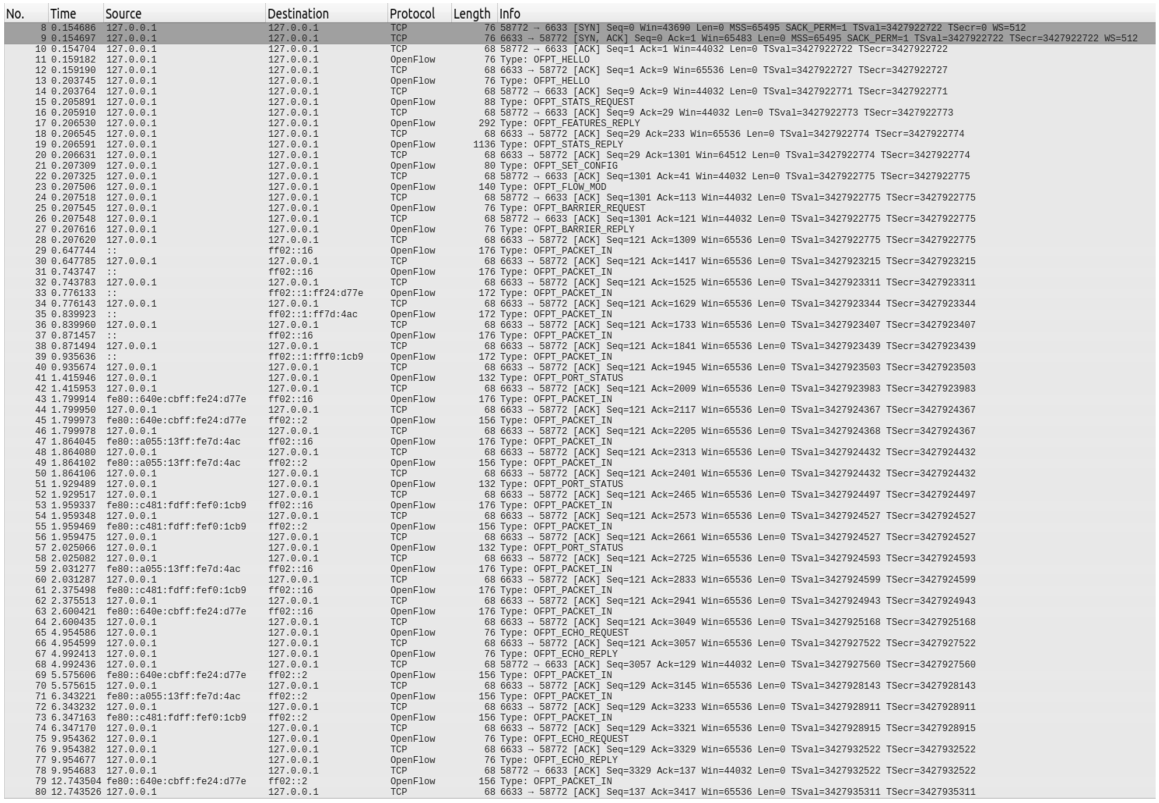
Figure 3.3. Script Execution Flowchart

3.5 The Data

3.5.1 Unencrypted Data

The data generated from the scenarios produce packet capture (PCAP) files. These files give a detailed look at what packets traversed the network. In this study we observe the data that traverses the control plane. This data is limited to packets that only span the connection between the controller and the switch. This allows the ability to determine what kind of network event occurred.

Figure 3.4 illustrates a full packet capture obtained from a scenario run with the unencrypted Mininet network. All packets have been discussed in detail in Section 3.1.1. Taking a detailed look into the capture files provides vital information regarding to the information passed from the controller to the switch and the switch to the controller. To start, the ports used are 58772 and 6633 for the switch and controller, respectively. This shows that only traffic being observed is solely from the control plane. We can also see the types of protocols used throughout the network. For this network, only the transmission control (TCP) and OpenFlow protocols are utilized. For the time column, this shows the time elapsed since the previous packet was sent. Getting more detailed in the information presented we can see the length of each packet.



No.	Time	Source	Destination	Protocol	Length	Info
8	0.154686	127.0.0.1	127.0.0.1	TCP	76	58772 → 6633 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=3427922722 TSecr=0 WS=512
9	0.154697	127.0.0.1	127.0.0.1	TCP	76	6633 → 58772 [SYN, ACK] Seq=0 Ack=1 Win=65483 Len=0 MSS=65495 SACK_PERM=1 TSval=3427922722 TSecr=3427922722 WS=512
10	0.154704	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=3427922722 TSecr=3427922722
11	0.159182	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_HELLO
12	0.159190	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=1 Ack=9 Win=65536 Len=0 TSval=3427922727 TSecr=3427922727
13	0.203745	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_HELLO
14	0.203764	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=9 Ack=9 Win=44032 Len=0 TSval=3427922771 TSecr=3427922771
15	0.205891	127.0.0.1	127.0.0.1	OpenFlow	88	Type: OFPT_STATS_REQUEST
16	0.205910	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=9 Ack=29 Win=44032 Len=0 TSval=3427922773 TSecr=3427922773
17	0.206530	127.0.0.1	127.0.0.1	OpenFlow	292	Type: OFPT_FEATURES_REPLY
18	0.206545	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=29 Ack=233 Win=65536 Len=0 TSval=3427922774 TSecr=3427922774
19	0.206591	127.0.0.1	127.0.0.1	OpenFlow	1136	Type: OFPT_STATS_REPLY
20	0.206631	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=29 Ack=1301 Win=64512 Len=0 TSval=3427922774 TSecr=3427922774
21	0.207309	127.0.0.1	127.0.0.1	OpenFlow	80	Type: OFPT_SET_CONFIG
22	0.207325	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=1301 Ack=41 Win=44032 Len=0 TSval=3427922775 TSecr=3427922775
23	0.207506	127.0.0.1	127.0.0.1	OpenFlow	140	Type: OFPT_FLOW_MOD
24	0.207518	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=1301 Ack=113 Win=44032 Len=0 TSval=3427922775 TSecr=3427922775
25	0.207545	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_BARRIER_REQUEST
26	0.207548	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=1301 Ack=121 Win=44032 Len=0 TSval=3427922775 TSecr=3427922775
27	0.207616	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_BARRIER_REPLY
28	0.207620	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1309 Win=65536 Len=0 TSval=3427922775 TSecr=3427922775
29	0.647744	::	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
30	0.647785	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1417 Win=65536 Len=0 TSval=3427923215 TSecr=3427923215
31	0.743747	::	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
32	0.743783	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1525 Win=65536 Len=0 TSval=3427923311 TSecr=3427923311
33	0.776133	::	ff02::1:ff24:d77e	OpenFlow	172	Type: OFPT_PACKET_IN
34	0.776143	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1629 Win=65536 Len=0 TSval=3427923344 TSecr=3427923344
35	0.839923	::	ff02::1:ff7d:4ac	OpenFlow	172	Type: OFPT_PACKET_IN
36	0.839960	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1733 Win=65536 Len=0 TSval=3427923407 TSecr=3427923407
37	0.871457	::	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
38	0.871494	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1841 Win=65536 Len=0 TSval=3427923439 TSecr=3427923439
39	0.935636	::	ff02::1:fff0:1cb9	OpenFlow	172	Type: OFPT_PACKET_IN
40	0.935674	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=1945 Win=65536 Len=0 TSval=3427923503 TSecr=3427923503
41	1.415946	127.0.0.1	127.0.0.1	OpenFlow	132	Type: OFPT_PORT_STATUS
42	1.415953	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2009 Win=65536 Len=0 TSval=3427923983 TSecr=3427923983
43	1.799914	fe80::640e:cbff:fe24:d77e	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
44	1.799950	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2117 Win=65536 Len=0 TSval=3427924367 TSecr=3427924367
45	1.799973	fe80::640e:cbff:fe24:d77e	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
46	1.799978	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2205 Win=65536 Len=0 TSval=3427924368 TSecr=3427924367
47	1.864045	fe80::a055:13ff:fe7d:4ac	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
48	1.864080	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2313 Win=65536 Len=0 TSval=3427924432 TSecr=3427924432
49	1.864102	fe80::a055:13ff:fe7d:4ac	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
50	1.864106	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2401 Win=65536 Len=0 TSval=3427924432 TSecr=3427924432
51	1.929489	127.0.0.1	127.0.0.1	OpenFlow	132	Type: OFPT_PORT_STATUS
52	1.929517	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2465 Win=65536 Len=0 TSval=3427924497 TSecr=3427924497
53	1.959337	fe80::c481:fdff:fef0:1cb9	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
54	1.959348	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2573 Win=65536 Len=0 TSval=3427924527 TSecr=3427924527
55	1.959469	fe80::c481:fdff:fef0:1cb9	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
56	1.959475	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2661 Win=65536 Len=0 TSval=3427924527 TSecr=3427924527
57	2.025066	127.0.0.1	127.0.0.1	OpenFlow	132	Type: OFPT_PORT_STATUS
58	2.025082	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2725 Win=65536 Len=0 TSval=3427924593 TSecr=3427924593
59	2.031277	fe80::a055:13ff:fe7d:4ac	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
60	2.031287	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2833 Win=65536 Len=0 TSval=3427924599 TSecr=3427924599
61	2.375498	fe80::c481:fdff:fef0:1cb9	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
62	2.375513	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=2941 Win=65536 Len=0 TSval=3427924943 TSecr=3427924943
63	2.600421	fe80::640e:cbff:fe24:d77e	ff02::16	OpenFlow	176	Type: OFPT_PACKET_IN
64	2.600435	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=3049 Win=65536 Len=0 TSval=3427925168 TSecr=3427925168
65	4.954586	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_ECHO_REQUEST
66	4.954599	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=121 Ack=3057 Win=65536 Len=0 TSval=3427927522 TSecr=3427927522
67	4.992413	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_ECHO_REPLY
68	4.992436	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=3057 Ack=129 Win=44032 Len=0 TSval=3427927560 TSecr=3427927560
69	5.575606	fe80::640e:cbff:fe24:d77e	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
70	5.575615	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=129 Ack=3145 Win=65536 Len=0 TSval=3427928143 TSecr=3427928143
71	6.343221	fe80::a055:13ff:fe7d:4ac	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
72	6.343232	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=129 Ack=3233 Win=65536 Len=0 TSval=3427928911 TSecr=3427928911
73	6.347163	fe80::c481:fdff:fef0:1cb9	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
74	6.347170	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=129 Ack=3321 Win=65536 Len=0 TSval=3427928915 TSecr=3427928915
75	9.954362	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_ECHO_REQUEST
76	9.954382	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=129 Ack=3329 Win=65536 Len=0 TSval=3427932522 TSecr=3427932522
77	9.954677	127.0.0.1	127.0.0.1	OpenFlow	76	Type: OFPT_ECHO_REPLY
78	9.954683	127.0.0.1	127.0.0.1	TCP	68	58772 → 6633 [ACK] Seq=3329 Ack=137 Win=44032 Len=0 TSval=3427932522 TSecr=3427932522
79	12.743504	fe80::640e:cbff:fe24:d77e	ff02::2	OpenFlow	156	Type: OFPT_PACKET_IN
80	12.743526	127.0.0.1	127.0.0.1	TCP	68	6633 → 58772 [ACK] Seq=137 Ack=3417 Win=65536 Len=0 TSval=3427935311 TSecr=3427935311

Figure 3.4. Full Packet Capture of Unencrypted OpenFlow Traffic

3.5.2 Encrypted Data

Noting the connection establishment for the encrypted network in Figure 3.5, just as with the unencrypted network, it begins with the TCP three-way handshake. From there we see the *Client Hello* message from the switch to the server. This message communicates the version of TLS that the switch is able to execute. Next is the *Server Hello* selecting the configurations of the cipher suite to use. For this scenario, TLS 1.3 is chosen. The switch then checks the configurations and generates the key to be used for the session. Once this three-step process is done the encrypted connection is established and all subsequent packets are encrypted.

15	0.009915	127.0.0.1	127.0.0.1	TCP	76	36238 → 6633 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=186223276 TSecr=0 WS=512
16	0.000009	127.0.0.1	127.0.0.1	TCP	76	6633 → 36238 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=186223276 TSecr=186223276 WS=512
17	0.000028	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=186223276 TSecr=186223276
18	0.000580	127.0.0.1	127.0.0.1	TLSv1.3	597	Client Hello
19	0.000016	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=1 Ack=530 Win=43520 Len=0 TSval=186223277 TSecr=186223277
20	0.001718	127.0.0.1	127.0.0.1	TLSv1.3	1742	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data
21	0.000017	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=530 Ack=1675 Win=42496 Len=0 TSval=186223278 TSecr=186223278
22	0.001327	127.0.0.1	127.0.0.1	TLSv1.3	1367	Change Cipher Spec, Application Data, Application Data, Application Data
23	0.000016	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=1675 Ack=1829 Win=43008 Len=0 TSval=186223280 TSecr=186223280
24	0.000028	127.0.0.1	127.0.0.1	TLSv1.3	98	Application Data
25	0.000003	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=1675 Ack=1859 Win=43008 Len=0 TSval=186223280 TSecr=186223280
26	0.001950	127.0.0.1	127.0.0.1	TLSv1.3	1219	Application Data
27	0.000015	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=2826 Win=43008 Len=0 TSval=186223282 TSecr=186223282

Figure 3.5. TLS Connection Establishment

Figure 3.6 displays a more generic message diagram associated with TLS 1.3 connection establishment.

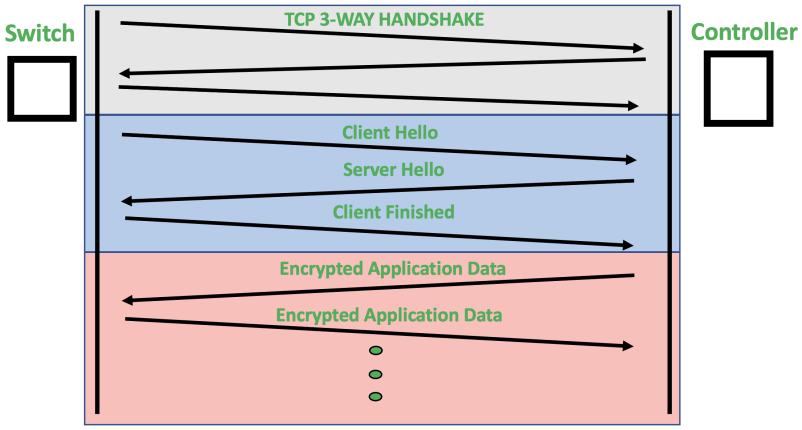
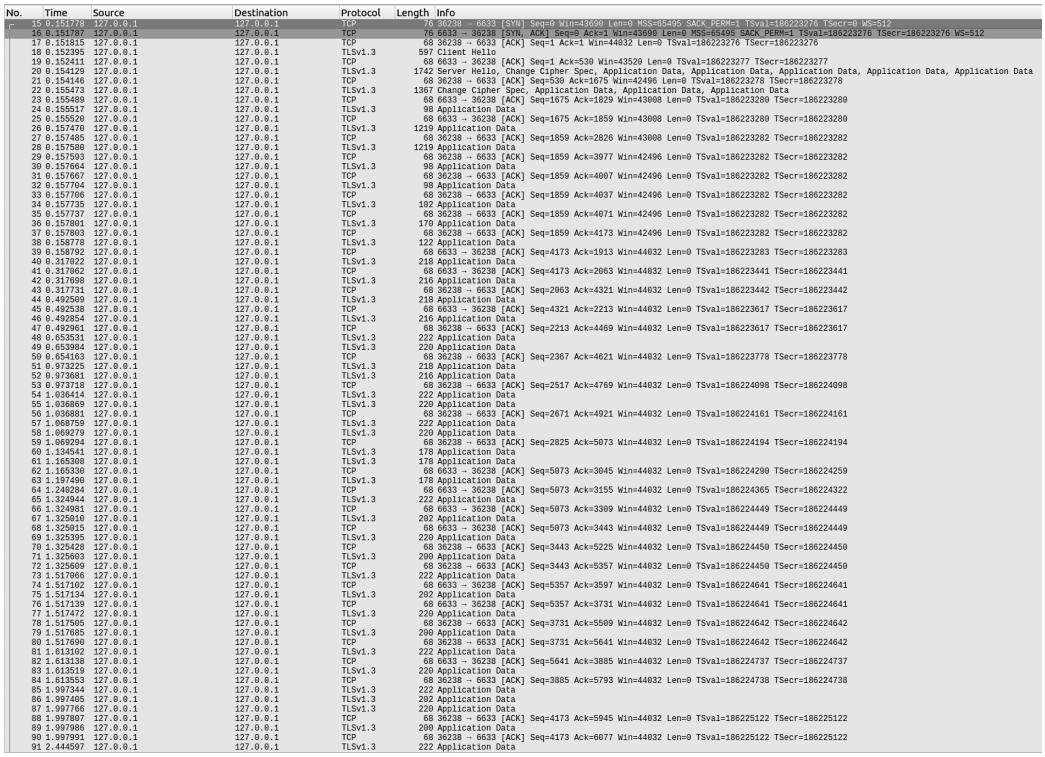


Figure 3.6. TLS 1.3 Message Diagram

Moving to the encrypted packets of the network data shown in Figure 3.7, it is immediately apparent there is not as much information provided by the PCAP file. Just as with the unencrypted file, the ports for the controller and switch are visible, as are the lengths of the packets. Protocols used were TLS 1.3 and TCP. The final bit of information provided is the time elapsed since the previous packet was sent. These packets, specifically the packets labeled application data, are the encrypted versions of the OpenFlow messages outlined in the previous section.



No.	Time	Source	Destination	Protocol	Length	Info
15	0.151778	127.0.0.1	127.0.0.1	TCP	76	36238 → 6633 [SYN] Seq=0 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=186223276 TSecr=0 WS=512
16	0.151787	127.0.0.1	127.0.0.1	TCP	76	6633 → 36238 [SYN, ACK] Seq=0 Ack=1 Win=43690 Len=0 MSS=65495 SACK_PERM=1 TSval=186223276 TSecr=186223276 WS=512
17	0.151815	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1 Ack=1 Win=44032 Len=0 TSval=186223276 TSecr=186223276
18	0.152395	127.0.0.1	127.0.0.1	TLSv1.3	597	Client Hello
19	0.152411	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=1 Ack=530 Win=43520 Len=0 TSval=186223277 TSecr=186223277
20	0.154129	127.0.0.1	127.0.0.1	TLSv1.3	1742	Server Hello, Change Cipher Spec, Application Data, Application Data, Application Data, Application Data, Application Data
21	0.154146	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=530 Ack=1675 Win=42496 Len=0 TSval=186223278 TSecr=186223278
22	0.155473	127.0.0.1	127.0.0.1	TLSv1.3	1367	Change Cipher Spec, Application Data, Application Data, Application Data
23	0.155489	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=1675 Ack=1829 Win=43008 Len=0 TSval=186223280 TSecr=186223280
24	0.155517	127.0.0.1	127.0.0.1	TLSv1.3	98	Application Data
25	0.155520	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=1675 Ack=1859 Win=43008 Len=0 TSval=186223280 TSecr=186223280
26	0.157470	127.0.0.1	127.0.0.1	TLSv1.3	1219	Application Data
27	0.157485	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=2826 Win=43008 Len=0 TSval=186223282 TSecr=186223282
28	0.157580	127.0.0.1	127.0.0.1	TLSv1.3	1219	Application Data
29	0.157593	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=3977 Win=42496 Len=0 TSval=186223282 TSecr=186223282
30	0.157664	127.0.0.1	127.0.0.1	TLSv1.3	98	Application Data
31	0.157667	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=4007 Win=42496 Len=0 TSval=186223282 TSecr=186223282
32	0.157704	127.0.0.1	127.0.0.1	TLSv1.3	98	Application Data
33	0.157706	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=4037 Win=42496 Len=0 TSval=186223282 TSecr=186223282
34	0.157735	127.0.0.1	127.0.0.1	TLSv1.3	102	Application Data
35	0.157737	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=4071 Win=42496 Len=0 TSval=186223282 TSecr=186223282
36	0.157801	127.0.0.1	127.0.0.1	TLSv1.3	170	Application Data
37	0.157803	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=1859 Ack=4173 Win=42496 Len=0 TSval=186223282 TSecr=186223282
38	0.158778	127.0.0.1	127.0.0.1	TLSv1.3	122	Application Data
39	0.158792	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=4173 Ack=1913 Win=44032 Len=0 TSval=186223283 TSecr=186223283
40	0.317022	127.0.0.1	127.0.0.1	TLSv1.3	218	Application Data
41	0.317062	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=4173 Ack=2063 Win=44032 Len=0 TSval=186223441 TSecr=186223441
42	0.317698	127.0.0.1	127.0.0.1	TLSv1.3	216	Application Data
43	0.317731	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=2063 Ack=4321 Win=44032 Len=0 TSval=186223442 TSecr=186223442
44	0.492509	127.0.0.1	127.0.0.1	TLSv1.3	218	Application Data
45	0.492538	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=4321 Ack=2213 Win=44032 Len=0 TSval=186223617 TSecr=186223617
46	0.492854	127.0.0.1	127.0.0.1	TLSv1.3	216	Application Data
47	0.492961	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=2213 Ack=4469 Win=44032 Len=0 TSval=186223617 TSecr=186223617
48	0.653531	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
49	0.653984	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
50	0.654163	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=2367 Ack=4621 Win=44032 Len=0 TSval=186223778 TSecr=186223778
51	0.973225	127.0.0.1	127.0.0.1	TLSv1.3	218	Application Data
52	0.973681	127.0.0.1	127.0.0.1	TLSv1.3	216	Application Data
53	0.973718	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=2517 Ack=4769 Win=44032 Len=0 TSval=186224098 TSecr=186224098
54	1.036414	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
55	1.036869	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
56	1.036881	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=2671 Ack=4921 Win=44032 Len=0 TSval=186224161 TSecr=186224161
57	1.068759	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
58	1.069279	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
59	1.069294	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=2825 Ack=5073 Win=44032 Len=0 TSval=186224194 TSecr=186224194
60	1.134541	127.0.0.1	127.0.0.1	TLSv1.3	178	Application Data
61	1.165308	127.0.0.1	127.0.0.1	TLSv1.3	178	Application Data
62	1.165330	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5073 Ack=3045 Win=44032 Len=0 TSval=186224290 TSecr=186224259
63	1.197490	127.0.0.1	127.0.0.1	TLSv1.3	178	Application Data
64	1.240284	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5073 Ack=3155 Win=44032 Len=0 TSval=186224365 TSecr=186224322
65	1.324944	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
66	1.324981	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5073 Ack=3309 Win=44032 Len=0 TSval=186224449 TSecr=186224449
67	1.325010	127.0.0.1	127.0.0.1	TLSv1.3	202	Application Data
68	1.325015	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5073 Ack=3443 Win=44032 Len=0 TSval=186224449 TSecr=186224449
69	1.325395	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
70	1.325428	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=3443 Ack=5225 Win=44032 Len=0 TSval=186224450 TSecr=186224450
71	1.325603	127.0.0.1	127.0.0.1	TLSv1.3	200	Application Data
72	1.325609	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=3443 Ack=5357 Win=44032 Len=0 TSval=186224450 TSecr=186224450
73	1.517066	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
74	1.517102	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5357 Ack=3597 Win=44032 Len=0 TSval=186224641 TSecr=186224641
75	1.517134	127.0.0.1	127.0.0.1	TLSv1.3	202	Application Data
76	1.517139	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5357 Ack=3731 Win=44032 Len=0 TSval=186224641 TSecr=186224641
77	1.517472	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
78	1.517505	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=3731 Ack=5509 Win=44032 Len=0 TSval=186224642 TSecr=186224642
79	1.517685	127.0.0.1	127.0.0.1	TLSv1.3	200	Application Data
80	1.517690	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=3731 Ack=5641 Win=44032 Len=0 TSval=186224642 TSecr=186224642
81	1.613102	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
82	1.613138	127.0.0.1	127.0.0.1	TCP	68	6633 → 36238 [ACK] Seq=5641 Ack=3885 Win=44032 Len=0 TSval=186224737 TSecr=186224737
83	1.613519	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
84	1.613553	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=3885 Ack=5793 Win=44032 Len=0 TSval=186224738 TSecr=186224738
85	1.997344	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data
86	1.997405	127.0.0.1	127.0.0.1	TLSv1.3	202	Application Data
87	1.997766	127.0.0.1	127.0.0.1	TLSv1.3	220	Application Data
88	1.997807	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=4173 Ack=5945 Win=44032 Len=0 TSval=186225122 TSecr=186225122
89	1.997986	127.0.0.1	127.0.0.1	TLSv1.3	200	Application Data
90	1.997991	127.0.0.1	127.0.0.1	TCP	68	36238 → 6633 [ACK] Seq=4173 Ack=6077 Win=44032 Len=0 TSval=186225122 TSecr=186225122
91	2.444597	127.0.0.1	127.0.0.1	TLSv1.3	222	Application Data

Figure 3.7. Full Packet Capture of Encrypted TLS Traffic

3.6 Data Labeling

Through data labeling, a small dataset was used to manually analyze packets attach meaning to the data. This process was tedious due to the inspection of each individual packet and in an attempt to find characteristics that might assist in identifying each packet or group of packets in the encrypted packet traces.

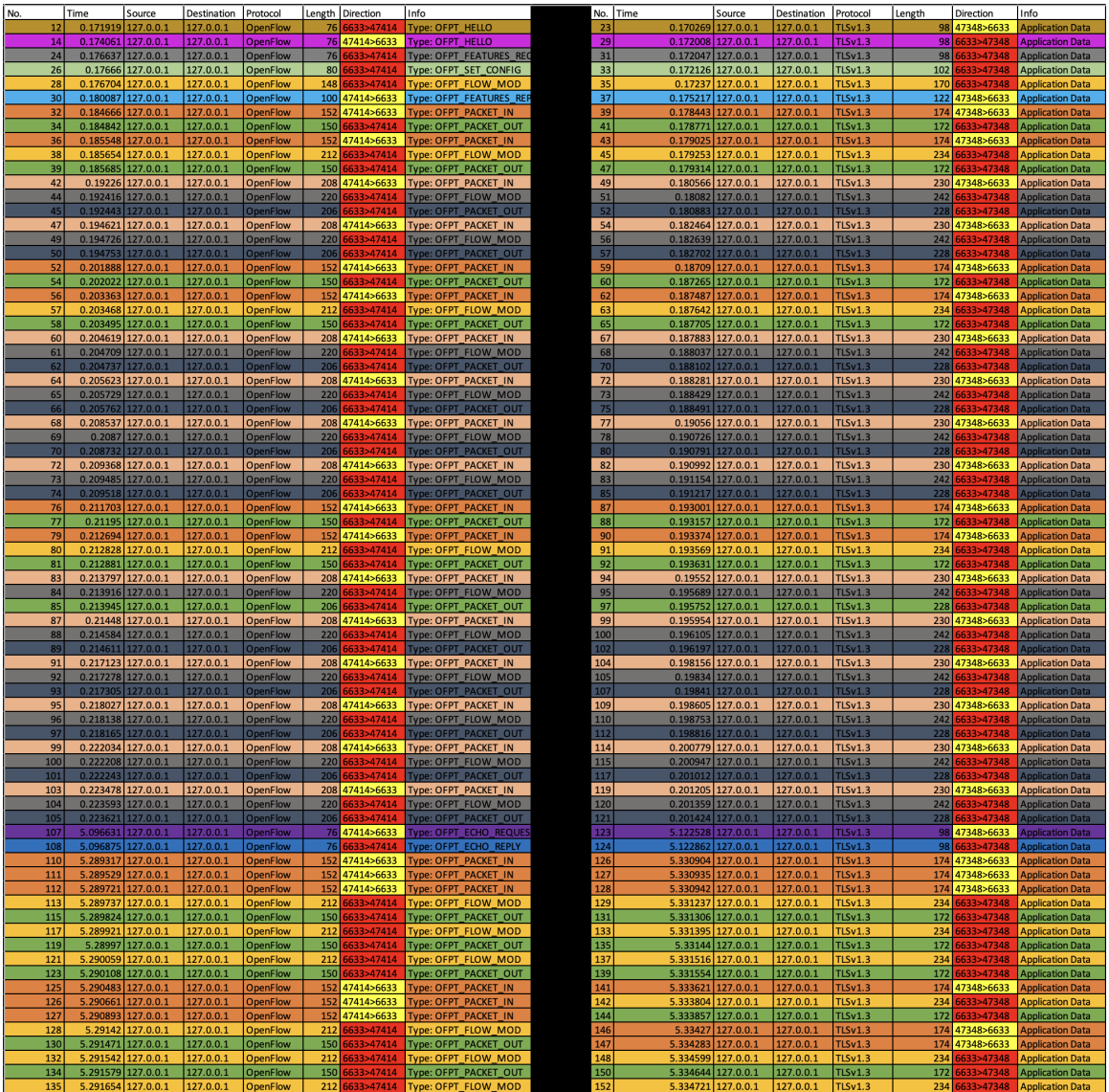
An initial intent of this study, the intent was to decrypt the TLS traffic to see the exact contents of each encrypted packet. This would provide 100% accurate labeling of each packet in an encrypted trace. However, after extensive research on how to achieve this with no results, along with time constraints, it was determined this was not possible.

A method that could be used is to utilize the pre-shared master secret created during the TLS secure connection establishment. This would allow for the decryption of packets encrypted during that specific session. Previous versions of TLS (TLS 1.0, 2.0, etc.) authorized the use of Rivest–Shamir–Adleman (RSA) key exchange to decrypt. This was not possible in TLS 1.3. Only Ephemeral Diffie Hellman key exchanges are authorized. Which is only

possible in a real-time live connection. There are solutions to work around this, but it was determined the method chosen would suffice for this study.

Data labeling was started by taking the two packet captures and attempting to marry up the transmitted packets. For example, identifying the unencrypted *OFPT_HELLO* message sent from the client to the sever and find where its equivalent message is in the encrypted packet capture file. This process was executed until all unencrypted packets had been mapped to a corresponding encrypted packet.

For the study, the TCP acknowledgement packet and the Internet Protocol version 6 (IPv6) packets were removed. The removal was necessary due to their trivial meaning with respect to network traffic inference. The result left only the Openflow messages. Figure 3.8 displays the output of just the Openflow messages.



No.	Time	Source	Destination	Protocol	Length	Direction	Info
12	0.171919	127.0.0.1	127.0.0.1	OpenFlow	76	6633>47414	Type: OFPT_HELLO
14	0.174061	127.0.0.1	127.0.0.1	OpenFlow	76	47414>6633	Type: OFPT_HELLO
24	0.176637	127.0.0.1	127.0.0.1	OpenFlow	76	6633>47414	Type: OFPT_FEATURES_REQ
26	0.17666	127.0.0.1	127.0.0.1	OpenFlow	80	6633>47414	Type: OFPT_SET_CONFIG
28	0.176704	127.0.0.1	127.0.0.1	OpenFlow	148	6633>47414	Type: OFPT_FLOW_MOD
30	0.180087	127.0.0.1	127.0.0.1	OpenFlow	100	47414>6633	Type: OFPT_FEATURES_REP
32	0.184666	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
34	0.184842	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
36	0.185548	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
38	0.185654	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
39	0.185685	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
42	0.19226	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
44	0.192416	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
45	0.192443	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
47	0.194621	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
49	0.194726	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
50	0.194753	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
52	0.201888	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
54	0.202022	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
56	0.203363	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
57	0.203468	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
58	0.203495	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
60	0.204619	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
61	0.204709	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
62	0.204737	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
64	0.205623	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
65	0.205729	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
66	0.205762	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
68	0.208537	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
69	0.2087	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
70	0.208732	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
72	0.209368	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
73	0.209485	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
74	0.209518	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
76	0.211703	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
77	0.21195	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
79	0.212694	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
80	0.212828	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
81	0.212881	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
83	0.213797	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
84	0.213916	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
85	0.213945	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
87	0.21448	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
88	0.214584	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
89	0.214611	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
91	0.217123	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
92	0.217278	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
93	0.217305	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
95	0.218027	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
96	0.218138	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
97	0.218165	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
99	0.222034	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
100	0.222208	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
101	0.222243	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
103	0.223478	127.0.0.1	127.0.0.1	OpenFlow	208	47414>6633	Type: OFPT_PACKET_IN
104	0.223593	127.0.0.1	127.0.0.1	OpenFlow	220	6633>47414	Type: OFPT_FLOW_MOD
105	0.223621	127.0.0.1	127.0.0.1	OpenFlow	206	6633>47414	Type: OFPT_PACKET_OUT
107	5.096631	127.0.0.1	127.0.0.1	OpenFlow	76	47414>6633	Type: OFPT_ECHO_REQUES
108	5.096875	127.0.0.1	127.0.0.1	OpenFlow	76	6633>47414	Type: OFPT_ECHO_REPLY
110	5.289317	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
111	5.289529	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
112	5.289721	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
113	5.289737	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
115	5.289824	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
117	5.289921	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
119	5.28997	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
121	5.290059	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
123	5.290108	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
125	5.290483	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
126	5.290661	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
127	5.290893	127.0.0.1	127.0.0.1	OpenFlow	152	47414>6633	Type: OFPT_PACKET_IN
128	5.29142	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
130	5.291471	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
132	5.291542	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
134	5.291579	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
135	5.291654	127.0.0.1	127.0.0.1	OpenFlow	212	6633>47414	Type: OFPT_FLOW_MOD
136	5.291687	127.0.0.1	127.0.0.1	OpenFlow	150	6633>47414	Type: OFPT_PACKET_OUT
138	10.096046	127.0.0.1	127.0.0.1	OpenFlow	76	47414>6633	Type: OFPT_ECHO_REQUES
139	10.096531	127.0.0.1	127.0.0.1	OpenFlow	76	6633>47414	Type: OFPT_ECHO_REPLY
141	15.096557	127.0.0.1	127.0.0.1	OpenFlow	76	47414>6633	Type: OFPT_ECHO_REQUES
142	15.096844	127.0.0.1	127.0.0.1	OpenFlow	76	6633>47414	Type: OFPT_ECHO_REPLY

No.	Time	Source	Destination	Protocol	Length	Direction	Info
23	0.170269	127.0.0.1	127.0.0.1	TLSv1.3	98	47348>6633	Application Data
29	0.172008	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>47348	Application Data
31	0.172047	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>47348	Application Data
33	0.172126	127.0.0.1	127.0.0.1	TLSv1.3	102	6633>47348	Application Data
35	0.17237	127.0.0.1	127.0.0.1	TLSv1.3	170	6633>47348	Application Data
37	0.175217	127.0.0.1	127.0.0.1	TLSv1.3	122	47348>6633	Application Data
39	0.178443	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
41	0.178771	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
43	0.179025	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
45	0.179253	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
47	0.179314	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
49	0.180566	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
51	0.18082	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
52	0.180883	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
54	0.182464	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
56	0.182639	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
57	0.182702	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
59	0.18709	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
60	0.187265	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
62	0.187487	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
63	0.187642	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
65	0.187705	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
67	0.187883	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
68	0.188037	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
70	0.188102	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
72	0.188281	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
73	0.188429	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
75	0.188491	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
77	0.19056	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
78	0.190726	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
80	0.190791	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
82	0.190992	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
83	0.191154	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
85	0.191217	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
87	0.193001	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
88	0.193157	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
90	0.193374	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
91	0.193569	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
92	0.193631	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
94	0.19552	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
95	0.195689	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
97	0.195752	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
99	0.195954	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
100	0.196105	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
102	0.196197	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
104	0.198156	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
105	0.19834	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
107	0.19841	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
109	0.198605	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
110	0.198753	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
112	0.198816	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
114	0.200779	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
115	0.200947	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
117	0.201012	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
119	0.201205	127.0.0.1	127.0.0.1	TLSv1.3	230	47348>6633	Application Data
120	0.201359	127.0.0.1	127.0.0.1	TLSv1.3	242	6633>47348	Application Data
121	0.201424	127.0.0.1	127.0.0.1	TLSv1.3	228	6633>47348	Application Data
123	5.122528	127.0.0.1	127.0.0.1	TLSv1.3	98	47348>6633	Application Data
124	5.122862	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>47348	Application Data
126	5.330904	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
127	5.330935	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
128	5.330942	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
129	5.331237	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
131	5.331306	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
133	5.331395	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
135	5.33144	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
137	5.331516	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
139	5.331554	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
141	5.333621	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
142	5.333804	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
144	5.333857	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
146	5.33427	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
147	5.334283	127.0.0.1	127.0.0.1	TLSv1.3	174	47348>6633	Application Data
148	5.334599	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
150	5.334644	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
152	5.334721	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>47348	Application Data
154	5.33476	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>47348	Application Data
156	10.123721	127.0.0.1	127.0.0.1	TLSv1.3	98	47348>6633	Application Data
157	10.124068	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>47348	Application Data
159	15.122549	127.0.0.1	127.0.0.1	TLSv1.3	98	47348>6633	Application Data
160	15.122897	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>47348	Application Data

Figure 3.8. Data Labeling

Starting at the bottom of the packet alignment, having observed the direction of the unencrypted packets and the size, there was a noticeable correlation. Unencrypted packets 138 through 142 align with encrypted packets 156 through 160. It was noticed that by identifying the size of the unencrypted packets, they were all the same (76 bytes). This same fixed-size characteristic was noticed with the encrypted packets as well (98 bytes). This is particularly

interesting because it showed TLS 1.3 adds 22 bytes of length to each packet. To further provide assurance these packets were the same content, we correlated them with their source-to-destination directionality (e.g. switch-to-controller or controller-to-switch) and found them to be the same. Therefore, encrypted packets 156 and 159 are *OFPT_ECHO_REQUEST* messages and encrypted packets 157 and 160 are *OFPT_ECHO_REPLY* messages with high certainty. This process was continued up the packet capture and unencrypted packets 132, 134, 135, and 136 correlated to encrypted packets 148, 150, and 152, and 154 respectively. Packets 132, 135, 148, and 152 are *OFPT_FLOW_MOD* messages and packets 134, 136, 150, and 154 were *OFPT_PACKET_OUT* messages.

Moving above these packets, it was observed some of the unencrypted packets are not in the same order as the encrypted packets. To understand this, attempts were made to find a packet to see if some packets were transmitted in a different order. For example, unencrypted packet 130 was transmitted from the controller to the switch and has a size of 150 bytes. Whereas the encrypted packet 147 was transmitted from the switch to the controller and has a size of 174. Using the heuristic, we know this does not follow the pattern. Using unencrypted packet 130 as the ground truth we attempt to find a packet that matches the direction transmitted and the size heuristic. We identified encrypted packet 144 as the corresponding packet to packet 130.

This method was continued for every packet. Once complete, we were able to identify each packet based on direction transmitted, size, and order. Figure 3.9 outlines the direction and lengths associated with each type of packet.

Type	Size	Specifics
Type: OFPT_HELLO	76	
Type: OFPT_FEATURES_REQUEST	76	
Type: OFPT_SET_CONFIG	80	
Type: OFPT_FLOW_MOD	148	
Type: OFPT_FEATURES_REPLY	100	
Type: OFPT_PACKET_IN	152	Arp Request (Switch to Controller)
Type: OFPT_PACKET_OUT	150	Arp Request (Controlle to Switch)
Type: OFPT_FLOW_MOD	212	
Type: OFPT_PACKET_IN	208	Ping Request/Reply (Switch to Controller)
Type: OFPT_FLOW_MOD	220	
Type: OFPT_PACKET_OUT	206	Ping Request/Reply (Controller to Switch)
Type: OFPT_ECHO_REQUEST	76	Switch to Controller
Type: OFPT_ECHO_REPLY	76	Controller to Switch

Type	Size	Specifics
Type: OFPT_HELLO	98	
Type: OFPT_FEATURES_REQUEST	98	
Type: OFPT_SET_CONFIG	102	
Type: OFPT_FLOW_MOD	170	
Type: OFPT_FEATURES_REPLY	122	
Type: OFPT_PACKET_IN	174	Arp Request (Switch to Controller)
Type: OFPT_PACKET_OUT	172	Arp Request (Controlle to Switch)
Type: OFPT_FLOW_MOD	234	
Type: OFPT_PACKET_IN	230	Ping Request/Reply (Switch to Controller
Type: OFPT_FLOW_MOD	242	
Type: OFPT_PACKET_OUT	228	Ping Request/Reply (Controller to Switch
Type: OFPT_ECHO_REQUEST	98	Switch to Controller
Type: OFPT_ECHO_REPLY	98	Controller to Switch

Figure 3.9. Packet Heuristics

3.7 Feature Identification

Moving to feature identification, this is the process of examining only the encrypted TLS data. With this data, features identified can assist in identifying OpenFlow packet types.

After examining the encrypted packets, the features that provided information were direction of transmission, packet length, and time. These three features provide a measurement property to differentiate individual packets during the learning and classification process. Source, destination, and protocol did not provide any substantial assistance to the classification algorithm due to all the packets having been sent to and from the same place and the protocol always being TLS 1.3. Also, packet number did not provide significant results during the study but may be used in the future to conduct further research. Unused features would be dropped in the data preprocessing section of the study to keep with the recommended techniques of machine learning. Figure 3.10 displays the output of the data prior to preprocessing.

No.	Time	Source	Destination	Protocol	Length	Direction	Info
23	0.195653	127.0.0.1	127.0.0.1	TLSv1.3	98	33122>6633	Application Data
29	0.196834	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
31	0.196881	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
33	0.196976	127.0.0.1	127.0.0.1	TLSv1.3	102	6633>33122	Application Data
35	0.197037	127.0.0.1	127.0.0.1	TLSv1.3	170	6633>33122	Application Data
37	0.197399	127.0.0.1	127.0.0.1	TLSv1.3	122	33122>6633	Application Data
39	4.682757	127.0.0.1	127.0.0.1	TLSv1.3	98	33122>6633	Application Data
41	4.683144	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
43	9.683231	127.0.0.1	127.0.0.1	TLSv1.3	98	33122>6633	Application Data
45	9.683622	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
47	14.683462	127.0.0.1	127.0.0.1	TLSv1.3	98	33122>6633	Application Data
48	14.683778	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
50	19.683099	127.0.0.1	127.0.0.1	TLSv1.3	98	33122>6633	Application Data
51	19.683578	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
53	22.480837	127.0.0.1	127.0.0.1	TLSv1.3	174	33122>6633	Application Data
54	22.481143	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>33122	Application Data
56	22.481413	127.0.0.1	127.0.0.1	TLSv1.3	174	33122>6633	Application Data
57	22.481589	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>33122	Application Data
59	22.481636	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>33122	Application Data
61	22.481954	127.0.0.1	127.0.0.1	TLSv1.3	206	33122>6633	Application Data
62	22.482277	127.0.0.1	127.0.0.1	TLSv1.3	250	6633>33122	Application Data
64	22.482324	127.0.0.1	127.0.0.1	TLSv1.3	204	6633>33122	Application Data
66	22.482638	127.0.0.1	127.0.0.1	TLSv1.3	206	33122>6633	Application Data
67	22.482831	127.0.0.1	127.0.0.1	TLSv1.3	250	6633>33122	Application Data
69	22.482878	127.0.0.1	127.0.0.1	TLSv1.3	204	6633>33122	Application Data
71	22.483137	127.0.0.1	127.0.0.1	TLSv1.3	198	33122>6633	Application Data
72	22.483298	127.0.0.1	127.0.0.1	TLSv1.3	250	6633>33122	Application Data
74	22.483349	127.0.0.1	127.0.0.1	TLSv1.3	196	6633>33122	Application Data
76	26.683852	127.0.0.1	127.0.0.1	TLSv1.3	98	33122>6633	Application Data
77	26.684225	127.0.0.1	127.0.0.1	TLSv1.3	98	6633>33122	Application Data
79	27.547789	127.0.0.1	127.0.0.1	TLSv1.3	174	33122>6633	Application Data
80	27.54826	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>33122	Application Data
82	27.548422	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>33122	Application Data
84	27.549299	127.0.0.1	127.0.0.1	TLSv1.3	174	33122>6633	Application Data
85	27.54982	127.0.0.1	127.0.0.1	TLSv1.3	234	6633>33122	Application Data
87	27.549898	127.0.0.1	127.0.0.1	TLSv1.3	172	6633>33122	Application Data

Figure 3.10. Features To Be Used

3.8 Classification Algorithms

Developing the classification algorithm requires observing both encrypted and unencrypted packets and identifying features that apply to both sets of data. Then utilizing those features to differentiate events that occur across the network.

Figure 3.11 illustrates splitting data in testing and training data sets. The training data set is used to train the classification algorithm. Accuracy can be increased by tuning certain parameters with the algorithm. The output of this is the classification model. Once the model is developed the testing data will be passed to the model to determine accuracy of the algorithm.

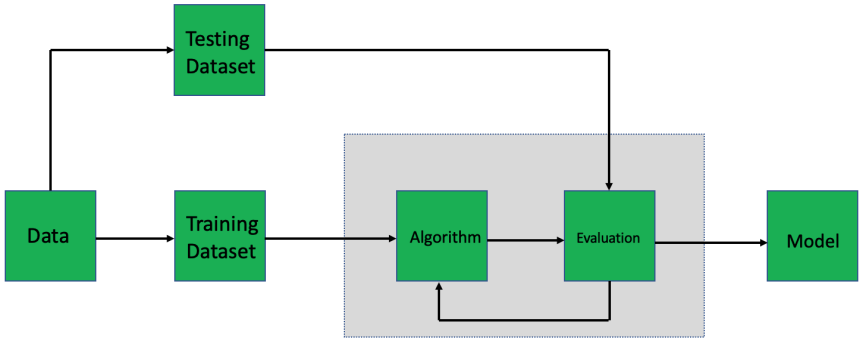


Figure 3.11. Classification Algorithm Development

For this study two classification algorithms were chosen, Decision Trees and Naive Bayes. Decision trees are a supervised machine learning algorithm that splits data according to specified parameters. The algorithm was chosen due to its ease of understanding and its prevalence when attempting to classify data. Naive Bayes is a probabilistic machine learning classifier. This specific classifier was chosen to provide contrast to the Decision Tree classifier for a broader range of study.

3.8.1 Preparing the Data

Before utilizing the data received from the network, it was necessary to prepare and clean the data. This step provided advantages when using the data to develop the classification algorithms.

First, the unnecessary columns that did not play a part in the classification algorithm were removed. Those columns were the packet number, source/destination IP address, and protocol. Next, processing of the data to provide better meaning for specific columns was needed. Time elapsed between each packet was used vice absolute time to better understand how long each packet took to be delivered. Since the direction format was not useful for the algorithm it was reformatted. Packets transmitted from the controller to the switch were designated a value of 0. And packets transmitted from the switch to the controller were designated a value of 1. This is known as a *numerical variable*. Finally, all the encrypted packets were provided a label developed from the unencrypted data explained in the data

labeling section. These labels were used during the training portion of the classification. Figure 3.12 shows a snippet of the results of the cleaning and preprocessing of the data.

Time	Length	Direction	Target
0.000037	98	1	Hello
0.000065	98	0	Hello
0.000038	98	0	Features Request
0.000035	102	0	Set Config
0.000057	170	0	Flow Mod
0.002127	122	1	Features Reply
1.020015	174	1	Packet In
0.000495	172	0	Packet Out
0.000805	174	1	Packet In
0.000341	234	0	Flow Mod
0.000074	172	0	Packet Out
0.00038	230	1	Packet In
0.000209	242	0	Flow Mod
0.000062	228	0	Packet Out
0.000314	230	1	Packet In
0.00022	242	0	Flow Mod
0.000062	228	0	Packet Out
0.002748	174	1	Packet In
0.000423	172	0	Packet Out

Figure 3.12. Features Used for Classification

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 4:
Results

This chapter outlines the results of the research conducted. It first reviews the dataset and how it was generated. Next the algorithm implementation and execution are covered. Finally the classification results and analysis are provided.

4.1 Dataset Preparation

The dataset shown in the previous chapter was generated from running 20 sessions of a network containing 1 controller, 1 switch, and 3 hosts. Each session would execute for 50 seconds and then shutdown. During this elapsed time, specific command would be given to generate different OpenFlow messages across the control plane of the SDN. Every run of the network would conduct a ping of all host of the network (*ping)*, write/read data through a TCP/UDP connection (*nc*), and retrieve a random web-page from the internet (*wget*). 3692 packets were generated from running the encrypted scenario with the aforementioned commands. After filtering out unneeded acknowledgment and IPv6 packets, only the OpenFlow messages remained. Table 4.1 outlines the statistics of the dataset utilized during classification.

Table 4.1. OpenFlow Dataset Statistics

OpenFlow Packet Type	Number of Packets Generated
Features Request	20
Set Config	20
Flow Mod	521
Features Reply	20
Packet In	560
Packet Out	562
Echo Request	134
Echo Reply	134
Total	2011

4.2 Algorithm Implementation

Decision Tree Classification

With the dataset cleaned and processed for classification, the Decision Tree algorithm was created. It was built through "a process known as binary recursive partitioning. This is an iterative method splitting the data into partitions, and then splitting it up further on each of the branches" [13].

The following is a high-level description of how I developed the Decision Tree classifier:

1. Loaded the dataset
2. Identified the features and the target variable
3. Divided the data into train and test data (80% training data and 20% test data)
4. Created the Decision Tree classifier
5. Trained the Decision Tree classifier
6. Created the confusion matrix
7. Created the accuracy report
8. Visualized the Decision Tree

Naive Bayes Classification

To provide a different method of classification, a Naive Bayes classifier was developed. "A Naive Bayes classifier assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature" [14]. The Naive Bayes classifier model took the features as an input and returned the probability of it being a specific type of message. Each feature is not dependent on any other, so the algorithm yielded different results as compared to the decision tree algorithm.

The Naive Bayes classifier development was similar to that of the Decision Tree, except there is no tree to be visualized. Here I:

1. Loaded the dataset
2. Identified the features and the target variable
3. Divided the data into train and test data (80% training data and 20% test data)
4. Created the Naive Bayes classifier

5. Trained the Naive Bayes classifier
6. Created the confusion matrix
7. Created the accuracy report

Each classifier was developed in Anaconda Navigator using Scientific Python Development Environment (SPYDER). This environment was chosen due to ease of use and familiarity from previous studies. The source code is provided in Appendix 1. The results and analysis of the classifications are provided in the Section 4.3.

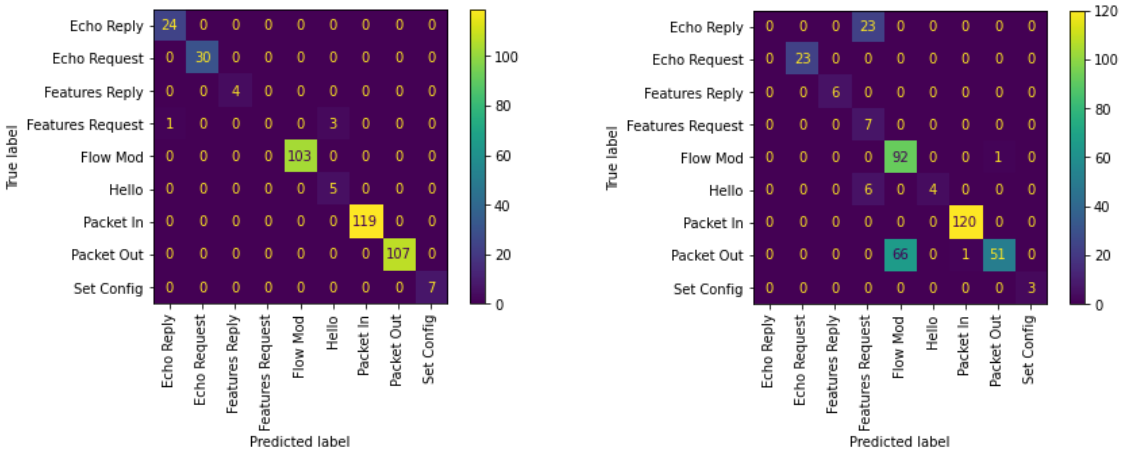
4.3 Classification Results

The classification results are shown in Table 4.2. The results were taken from the average of running the classification algorithm 15 times. The Decision Tree algorithm performed quite well, returning an accuracy of 99% and a n F-1 score of 98%. This is due to the method of splitting the data each time based on a specified parameter. As displayed, the Naive Bayes did not return optimal results. It returned an accuracy of 75% and an F-1 score of 74%. This is due to the features being independent from one another. Therefore features would not be used together to classify each packet.

Table 4.2. Results of Classification Algorithms.

Classification Algorithm	**Total number of packets**	**Packets classified**	**Accuracy**	**F-1 Score**
1: Naive Bayes	2011	403	75%	74%
2: Decision Tree	2011	403	99%	98%

When examining the confusion matrices on Figure 4.1, a more thorough explanation of how the classifiers identified each packet is given.



(a) Decision Tree Confusion Matrix

(b) Naive Bayes Confusion Matrix

Figure 4.1. Confusion Matrix Results

The diagonal plot shows the correct classification of each packet. Anything under the diagonal represents a packet that was the correct packet, but the model classified it incorrectly (false negative). Anything above the diagonal represents a packet that was wrong, but the classifier labeled it as the correct type of packet (false positive). Within the Decision Tree confusion matrix we see the *Features_Request* message misidentified. One of the packets was the correct type of packet but it was labeled as an *Echo_Reply* message. And three others were wrong (they were actually *Hello* messages) but it labeled them as a correct *Features_Request* message. In both instances the training of the classifier plays a significant role on why this occurs. Given that the *Hello*, *Features_Request*, and *Echo_Reply* packets all have the similar feature values it is easy to understand how the classifier might mis-label them.

For the Naive Bayes confusion matrix, the logic is the same. Except in this classifier *Echo_Reply* packets were predicted to be *Features_Request* packet. Then *Packet_Out* packets are misidentified as *Flow_Mod* packets. Since both sets of packets have similar values and Naive Bayes treats features independently the misidentification is higher with this classifier.

Further analysis was conducted in the area of features. This analysis was conducted to ensure the accuracy results were valid and did not memorize the data during training. The

accuracy was a result of utilizing each feature (time, length, direction) individually to see how well the classifier performed. Table 4.3 displays the results.

Table 4.3. Feature Importance Analysis

Classification Algorithm	**Time Feature**	**Direction Feature**	**Length Feature**	**Overall Accuracy**
1: Naive Bayes	22%	25%	26%	75%
2: Decision Tree	40%	54%	85%	98%

As the table shows, the individual features utilized in the decision tree classifier yielded higher classification accuracy, therefore leading to a higher accuracy when all are used together. What is particularly interesting about the Naive Bayes classifier's features is that all yielded similar accuracy. This confirms the fact that Naive Bayes classifiers treat each feature independently.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 5:
Conclusion and Future Work

5.1 Conclusions

The primary hypothesis of the thesis introduced in Section 1.3: *Identifying and classifying encrypted TLS data from SDNs to infer network events will provide an effective method of identifying potential network vulnerabilities and attacks*, is positively answered by the completion of gathering, labeling, and classifying encrypted OpenFlow packets within a Software Defined Network. The ability to infer the control packet type will allow for a more accurate monitoring of what is occurring on the network.

In addition, the research conducted was able to answer two of the three research questions posed in Section 1.3:

- First, *does selecting a specific classification algorithm impact inference accuracy?* The simple answer based on results presented in Chapter 4 is *YES*. Decision Tree is shown to perform much better than Naive Bayes.

- Second, *does the size and topology of the network affect the accuracy and speed of the classification?* If more data is generated from a bigger network, then will in fact take longer to classify the increased number of packets. However, both Decision Tree and Naive Bayes algorithms are able to scale to large datasets. Given the accuracy of the current algorithms, it is only logical with more data to train on, the accuracy would therefore increase.

- Finally, and unfortunately the final research question: *will the classification solution be deployable in Navy platforms* was not able to be researched. Initial thoughts would be that it is possible, but further work would be needed to build a testbed that is able to simulate a variety of Navy platforms. Not having the capability of gathering unencrypted data for labeling and then also being able to generate the equivalent encrypted data for classification purposes halted this process.

In summary, the results indicate that it *is* possible to infer encrypted TLS network events within software defined networks. Classification accuracy depends on algorithm selection. Results show a high of 99% accuracy when identifying encrypted packets. This study is easily repeatable and can be streamlined for optimization for future use.

5.2 Recommendations for Future Work

The work involving the gathering, labeling, and classifying of the data solution revealed a number of areas on which future research efforts could focus to advance and evolve the research and the results described in this thesis. These following potential areas are listed in no particular order:

- Further research could be placed in the area of other network protocols. This study utilized OpenFlow as the protocol to decipher messages. The expansion of other protocols would assist network monitors and managers in identifying malicious attacks conducted from other services introduced in the network.

- Effort could also be placed in the area of live classification. Testing conducted during this study occurred after data collection was complete. Live classification could be implemented within Navy networks to enhance network management and oversight. Having the ability to capture encrypted packets in real-time flowing across a network would provide substantial benefits to organizations wanting to see network content.

- Different classification algorithms could be an area of future work as well. Only two classification algorithms we used during this study and returned decent to great results. With the future search in algorithms, there could also be more research placed on the generation of different features. Since three features were used, there are quite a few more that could return promising results.

- The decryption of TLS 1.3 traffic is this final area of future work. There are methods of decrypting TLS 1.3 traffic available. The ability to decrypt said traffic provides more 100% accuracy in labeling data, therefore increased accuracy in the classifica-

tion. More time and research are needed to accomplish this task.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: Source Code for Classification Algorithms

The follow source code was utilized for a Decision Tree Classifier:

```
#Import necessary libraries
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
from sklearn.metrics import plot_confusion_matrix
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt

# Label Columns
col_names = ['Time', 'Length', 'Direction', 'Target']
tar_name = ['Target']
# Load dataset
df = pd.read_csv('total_data.csv')

#%%
# Identify Feature and Target Variable
feature_cols = ['Time', 'Length','Direction']
X = df[feature_cols] # Features
y = df.Target # Target variable

#%%
# Split dataset into training set and test set 80% training and 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
# Create Decision Tree classifer object
clf = DecisionTreeClassifier(max_features=3,max_depth=6)

# Train Decision Tree Classifer
clf = clf.fit(X_train,y_train)
```

```
y_pred = clf.predict(X_test)
# Plot Confusion Matrix
plot_confusion_matrix(clf, X_test, y_test, xticks_rotation=90)
# Calculate Feautre Importance
feat_importance = clf.tree_.compute_feature_importances(normalize=False)
# Print Results
print(classification_report(y_test,y_pred))
print(accuracy_score(y_test, y_pred))
print("Feat Importance = " + str(feat_importance))

#%%
# Visualize Decision Tree
plt.figure(figsize=(60,25))
plot_tree(clf,
          feature_names = feature_cols,
          class_names = df.Target,
          filled = True,
         proportion= True,
         fontsize=10,
                    rounded = True )

plt.savefig('tree_visualization.png')
```

The follow source code was utilized for a Naive Bayes Classifier:

```
#Import necessary libraries
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix
from sklearn.metrics import plot_confusion_matrix
from sklearn.naive_bayes import GaussianNB
from sklearn.inspection import permutation_importance

# Label Columns
col_names = ['Time', 'Length', 'Direction', 'Target']
tar_name = ['Target']
# Load dataset
df = pd.read_csv('total_data.csv')

#%%
# Identify Feature and Target Variable
feature_cols = ['Time', 'Length','Direction']
X = df[feature_cols] # Features
y = df.Target # Target variable

# Split dataset into training set and test set 80% training and 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state = 0, test_size=0.2)

# Train Naive Bayes classifier
gnb = GaussianNB().fit(X_train, y_train)
gnb_predictions = gnb.predict(X_test)
# Plot Confusion Matrix
plot_confusion_matrix(gnb, X_test, y_test, xticks_rotation=90)

y_pred = gnb.predict(X_test)

accuracy = gnb.score(X_test, y_test)
# Print Results
```

```
print(accuracy)
print('################################################')
print(classification_report(y_test,y_pred))
print('################################################')
# creating a confusion matrix
print(confusion_matrix(y_test, gnb_predictions))

print("Number of mislabeled points out of a total %d points : %d"
      % (X_test.shape[0], (y_test != y_pred).sum()))

imps = permutation_importance(gnb, X_test, y_test)
print(imps.importances_mean)
```

List of References

[1] M. Liberatore and B. N. Levine, “Inferring the source of encrypted http connection,” M.S. thesis, Dept. of Computer Science, University of Massachusetts, Amherst, MA, USA, 2006.

[2] P. Menuka, K. Piamrat, and S. Hamma, “Network traffic classification using machine learning for software defined networks,” Dept. of Computer. Science, University of Nantes, Tech. Rep. 17-59, 2019 [Online]. Available: https://hal.archives-ouvertes.fr/hal-02379020

[3] The World Bank. United Nations Population Division. [Online]. Available: http://data.worldbank.org/indicator/SP.DYN.LE00.FE.IN. Accessed Dec. 10, 2020.

[4] M. Sjoholmsierchio, “Software defined networks: Protocol dialects,” M.S. thesis, Dept. of Computer Science, Naval Postgraduate School, Monterey, CA, USA, 2019.

[5] AVI Networks. "TLS Proxy," Aug. 30, 2020. [Online]. Available: https://avinetworks.com/glossary/tls-proxy/

[6] T. Dierks, “The transport layer security protocol version 1.1,” Internet Requests for Comments, 04 2006. Available: https://tools.ietf.org/html/rfc4346

[7] T. Dierks, “The transport layer security protocol version 1.2,” Internet Requests for Comments, 08 2008. Available: https://tools.ietf.org/html/rfc5246

[8] E. Rescorla, “The transport layer security protocol version 1.3,” Internet Requests for Comments, 08 2018. Available: https://tools.ietf.org/html/rfc8446

[9] A. Verma, “Study and evaluation of classification algorithms in data mining,” *International Research Journal of Engineering and Technology*, vol. 5, no. 8, August 2018. [Online]. https://www.irjet.net/archives/V5/i8/IRJET-V5I8223.pdf.

[10] C. V. Wright, F. Monrose, and G. M. Masson, “On inferring application protocol behaviors in encrypted network traffic,” *Journal of Machine Learning Research*, vol. 7, no. 100, December 2006. [Online]. doi: https://www.jmlr.org/papers/volume7/wright06a/wright06a.pdf.

[11] A. Rasteh, F. Delpech, C. Aguilar-Melchor, R. Zimmer, S. B. Shouraki, and T. Masquelier, “Encrypted internet traffic classification using a supervised spiking neural network,” M.S. thesis, Cornell University, Ithaca, NY, USA, 2021.

[12] ONF, *OpenFlow Switch Specification*, ONF, Open Networking Foundation, March 2015. [Online]. Available: https://opennetworking.org/wp-content/uploads/2014/10/openflow-switch-v1.5.1.pdf

[13] A. Chakure, "Decision tree classification," Medium, July 5, 2019, [Online]. Available: https://medium.com/swlh/decision-tree-classification-de64fc4d5aac

[14] S. Ray. 6 Easy Steps to Learn Naive Bayes Algorithm with codes in Python and R, Sep. 11, 2017. [Online]. Available: https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California