8-23-2022 2:30 PM

Anonymization & Generation of Network Packet Datasets Using Deep learning

Spencer K. Vecile, *The University of Western Ontario*

Supervisor: Samarabandu, Jagath, *The University of Western Ontario*

A thesis submitted in partial fulfillment of the requirements for the Master of Engineering Science degree in Electrical and Computer Engineering

Abstract

Corporate networks are constantly bombarded by malicious actors trying to gain access. The current state of the art in protecting networks is deep learning-based intrusion detection systems (IDS). However, for an IDS to be effective it needs to be trained on a good dataset. The best datasets for training an IDS are real data captured from large corporate networks. Unfortunately, companies cannot release their network data due to privacy concerns creating a lack of public cybersecurity data. In this thesis I take a novel approach to network dataset anonymization using character-level LSTM models to learn the characteristics of a dataset; then generate a new, anonymized, synthetic dataset, with similar characteristics to the original. This method shows excellent performance when tested for characteristic preservation and anonymization performance on three datasets. One that includes malicious and benign URLs, one with DNS packets, and one with malicious and benign TCP packets. Using this method I take the first step in solving the lack of publication of private network datasets.

Keywords: Long Short-Term Memory Networks, Anonymization, Cybersecurity, Deep learning, Dataset Anonymization, Synthetic Data, Network Dataset Anonymization

Summary for Lay Audience

Corporate networks are constantly bombarded by hackers trying to gain access. The current state of the art in protecting networks is using artificial intelligence (AI) driven intrusion detection systems (IDS). However, for an IDS to be effective it needs to learn what a hacker's network activity looks like from a good dataset. The best datasets for training an IDS are real and from large corporate networks. Unfortunately, companies cannot release their network data due to privacy concerns creating a lack of public cybersecurity data. In this thesis I take a novel approach to network dataset anonymization using AI to learn the characteristics of a dataset; then generate a new, anonymized, synthetic dataset, with similar characteristics to the original. This method is tested for characteristic preservation and anonymization performance on three datasets. One that includes malicious and benign website addresses, one with DNS packets, and one with malicious and benign TCP packets. The results showed the AI was able to learn the structure and composition of these datasets and then generate its own synthetic anonymized version of these datasets. Using this AI-driven approach I take the first step in solving the lack of publicly available private network datasets for training IDSs.

Acknowlegements

First and foremost I would like to thank my supervisor Dr. Jagath Samarabandu for his continuous guidance and support throughout my research. He consistently allowed me the freedom to make this project my own but steered me in the right direction whenever I needed it. His knowledge in the process of taking a thesis project from concept to reality really has been indispensable.

I would also like to thank my parents for their encouragement, advice, and financial support. None of this would have been possible without the two of you. To my mom, Susan Keen, you are always there when I need you for emotional support and guidance. To my dad, Dino Vecile thank you for setting the example of the man I always aspire to become. Your drive to be the best in whatever you do motivated me every day throughout this process and kept me going even when I was down. In addition, I would like to thank you for always helping me with my writing throughout my entire university career (even when it was super last minute).

Contents

List of Figures

List of Tables

List of Appendices

List of Abbreviations, Symbols, and Nomenclature

CNN	Convolutional Neural Network
CSV	Comma Separated Value
DDoS	Distributed Denial of Service
DoS	Denial of Service
GAN	Generative Adversarial Network
HIDS	Host Intrusion Detection System
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IDS	Intrusion Detection System
IP	Internet Protocol
JSON	JavaScript Object Notation
LSTM	Long Short-Term Memory
NIDS	Network Intrusion Detection System
RAM	Random Access Memory
SQL	Structured Query Language

SSD	Solid-State Drive
t-SNE	T-Distributed Stochastic Neighbor Embedding
tanh	Hyperbolic Tangent
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UID	Unique Identifier
UMAP	Uniform Manifold Approximation and Projection
XSS	Cross-Site Scripting

Chapter 1

Introduction

1.1 Motivation

When doing machine learning research or applying it to a problem, there is always one constant, regardless of the project or application; a need for vast amounts of data. This acquisition of data is a challenging process both in terms of time and resources for many reasons, but mainly because data contains valuable information that is not often given away for free. This case is especially true for private companies whose data can contain trade secrets, or they may be in the business of data like social media companies. If they were to publicize these datasets first, they would have to ensure all private and privileged information was removed by having a data analyst go over the entire dataset. Which for a large dataset with billions of data points is just not feasible or would be incredibly costly. Now think about a world where these datasets could be anonymized not by a data analyst but by a deep learning algorithm. This is the goal of my research, and specifically, I will be focusing on cybersecurity datasets used to train intrusion detection systems (IDS). Please see Section 2.3 for a high-level description of IDS and the data needed to train such an IDS.

In the past, companies used rule-based network intrusion detection systems [13] to detect attacks on their network. While rules are still an integral part of IDSs, an IDS can be im-

proved by applying deep learning to this task [9]. This means that now the rules are not only defined by human experience but also a deep learning models learned parameters. For that model to be effective it must be trained on a good dataset. A good dataset has the following three characteristics: up to date, labeled, and contains realistic user behavior [24]. Realism is especially important as experimental results have shown that classification performance is significantly reduced when an IDS is used in a network environment that is significantly different than where its training data was extracted [14]. This should not be a problem as there is a lot of cybersecurity data globally, but most of it is held by private companies that are unwilling to release it for confidentiality reasons [16]. Unfortunately, this lack of data creates a dire need for more data for cybersecurity research and deep learning applications. When companies are willing to give access to their network data, the data is most of the time heavily anonymized [20].

This is a significant problem because to train and evaluate supervised learning algorithms, high-quality, realistic datasets are required. This thesis proposes using character-level long short-term memory (LSTM) models [8] to learn the characteristics of a private, real dataset. Then, generate a new, synthetic dataset that is characteristically indistinguishable from the original data it replaces; while being different enough to be considered anonymized. Using this method, the new synthetic data can be publicized without compromising the privacy of the original data provider. Allowing researchers to train and evaluate their IDSs using this publicly available synthetic data.

In another case, think about a company that wants to outsource the development of an IDS to protect their corporate network. They have a few options. One possible option is to hand over all their network data and spend a large amount of resources on legal and enforcement activities to ensure confidentiality is secured legally and enforced sufficiently. This, however, may not always be possible depending on the nature of their business. The second option would be to heavily anonymize the dataset themselves, which would take a huge amount of man-hours and resources. This will also probably strip the data of a lot of its most useful features for an

IDS. The final option for the company creating the IDS for them would be to use an already publicly available dataset to train the model. The problem with this approach is that when you train and tune a model on one set of data and deploy it on another, you will probably get much worse performance, especially when that data is from a completely different networking environment [14].

In the situation mentioned above, several weaknesses with current methods of sanitizing data were presented. Motivated by the aforementioned problems, this thesis proposes a deep learning-based solution. How this solution overcomes the above weaknesses is presented below.

Using the method of anonymization presented in this thesis mitigates the confidentiality problem, through the anonymization factor, and synthetic nature of the new dataset, avoiding the need of resources for legal and enforcement activities. It solves the second problem since the dataset is anonymized in-house, but instead of using a large amount of resources by doing it manually, it is done less labour-intensively by a deep learning model. Lastly, it solves the third problem since the dataset will be characteristically similar to the original. So, when the finished intrusion prevention system is deployed on the company's network, the data it was trained on came from their specific network architecture, making the performance of the model similar in training and deployment.

1.2 Contributions

- A novel application of character-level LSTM models for anonymization and generation of packet-level network data that can be used for the training and evaluation of intrusion detection systems. The model is first trained on the real data so it can learn the characteristics of the data. Then, it generates a new synthetic dataset, and through information loss, during the generation process, the original dataset is anonymized. The goal of this is to create a system that can allow for the publication of private network datasets. This

addresses a lack of high-quality realistic datasets that come from large corporate networks. To my knowledge, anonymization and generation of packet-level cybersecurity datasets using character-level LSTM models has never been attempted.

- Implementation of character-level LSTM model designed to train on batched, variable-length URLs and DNS packets. Once trained on this data, it can generate variable length URLs and DNS packets of its own. The ability of the LSTM to control the length and composition of these URLs is given by the addition of start and end tokens.

- Implementation of character-level LSTM model designed to learn the characteristics of malicious and benign TCP packets. Once trained on this data, it can generate synthetic anonymized malicious and benign TCP packets.

- Implementation of batch shuffling technique for training that can handle both variable and fixed-length sequences. Resulting in saved time and less overfitting in the models.

- Evaluation of anonymization performance of LSTM models on URL, DNS, and TCP datasets. Performance is evaluated using the Levenshtein similarity ratio for all three datasets. The Levenshtein ratio measures the difference between the real and synthetic datasets. However, before this method could be used in production an actual measure of privacy would have to be performed such as differential privacy [6], but that is left to future work. Anonymization performance is also reinforced visually by evaluating graphs of Uniform Manifold Approximation and Projection (UMAP) dimensionality reduction.

- Implementation of multi-threaded Levenshtein similarity ratio calculation. This was shown to significantly improve the computational efficiency when comparing the synthetic data, to the real data, to determine the anonymization performance of the models.

- Evaluation of the ability of the LTSM model to conserve the characteristics of the original data in the synthetic datasets. Characteristic preservation was evaluated on the URL and TCP datasets by training two LSTM classifiers. One trained on only real data and then

tested on a small, never before seen test set of the real data and the entire synthetic dataset. The second was trained only on synthetic data and then tested on a small, never before seen test set of the synthetic data and the entire real dataset. To reinforce the findings of the classifier, the TCP dataset was also evaluated using UMAP dimensionality reduction to confirm characteristic preservation visually. Since the DNS dataset has no malicious part in it, evaluating it with a classifier is not possible, so UMAP was used to evaluate characteristic preservation.

- Implementation of character-level LSTM model designed to classify fixed or variable length sequences in a batched manner. The model had excellent results on all datasets it was evaluated on.

- Implementation of a method to transfer the flow-based labels of CIC-IDS2017 [24] to the packet-level data that came with it.

- Creation of anonymized versions of the URL, DNS, and CIC-IDS2017 [24] datasets.

Parts of Chapter 4 were published [28]. The research was done solely by me, but the published paper was based on a paper Mr. Lacroix and I wrote about the research for a class. However, before publication, the paper was completely redone by myself with editing assistance from Dr. Grolinger and Dr. Samarabandu.

1.3 Thesis Outline

The organization of the rest of this thesis is the following: Chapter 2 is the background, which describes the basics of LSTM networks in Section 2.1, the formats of network data capture in Section 2.2 and an overview of intrusion detection systems in Section 2.3.

Chapter 3 contains the related work. Since no direct comparison can be made to my method, it is split up into other anonymization techniques in Section 3.1 and network data generation techniques in Section 3.2.

Chapters 4, 5, and 6 are the main body of the thesis and are laid out in such a way that they show the progression of my research from easiest to most difficult problem. As a starting point, it was decided to see if an LSTM could generate a synthetic dataset that contained malicious and benign URLs. Once that was confirmed possible, the next step was to see if the LSTM model could generate a synthetic packet-level DNS dataset. This was chosen as the next (slightly more difficult) step since DNS packets contain URLs in the payload. Once this was confirmed possible, I moved on to the most complicated part, which was generating and anonymizing a packet-level TCP dataset.

More formally, Chapter 4 contains malicious and benign URL dataset generation and anonymization. Chapter 5 contains DNS packet generation and anonymization with LSTM network hyperparameter tuning. Chapter 6 contains the main goal of my research which is malicious and benign TCP dataset generation and anonymization.

Finally, in Chapter 6, conclusions and future work will be discussed.

Chapter 2

Background

This chapter will go over background knowledge that will be helpful in understanding the rest of this thesis. First, LSTM networks will be described as they are heavily used in the synthetic data generation process and the evaluation of the synthetic datasets. Second, different formats of network data capture will be described as this will be helpful information when understanding the dataset used in Chapter 6.

2.1 Long Short-Term Memory (LSTM) Networks

LSTM networks are a type of recurrent neural network (RNN) that was designed as an upgrade to generic RNNs to overcome the vanishing/exploding gradient problem [27]. LSTM networks have recurrent connections, which makes them specifically suited to process sequential data. These recurrent connections give the network memory since during each step, data from the previous step and all steps before it is fed back into the network. The heart of an LSTM network is the LSTM cell shown in Fig. 2.1.

There are three main trainable parts to the LSTM cell. The forget gate, the input gate, and the output gate. These three parts, combined with input data, the cell state, and the hidden state, make up the LSTM unit. The cell state acts as an information highway that transports relevant information down the sequence chain. Think of it as the memory of the network. This

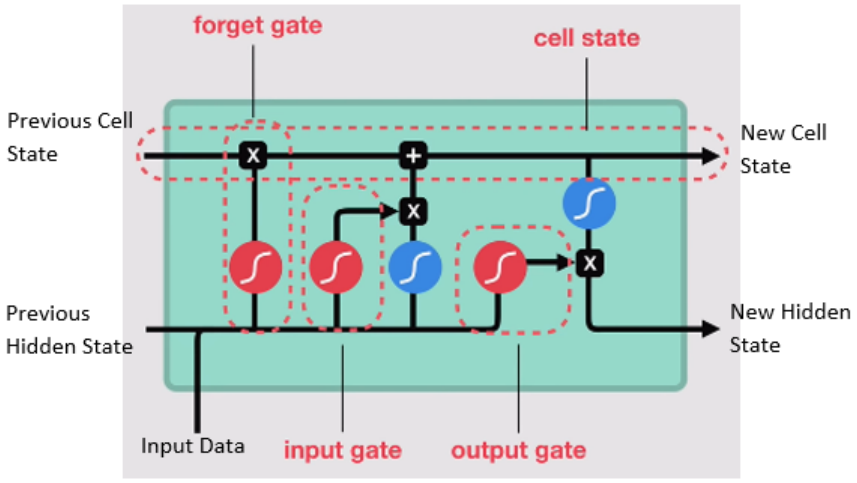

Figure 2.1: Internals of an LSTM cell

theoretically could carry information all the way from the first step to the last, and on its way, information gets added or removed from the cell state via gates. The other state is the hidden state which is the output from a previous time step and is combined with new input data during training. Once the input data and the hidden state are combined into a vector, the first gate they pass is the forget gate.

The forget gate decides which bits of the cell state should be remembered and which should be forgotten, given both the previous hidden state and the new input data. To do this, the hidden state and input data are put through a neural network that generates a vector where each element is constrained to between zero and one by the sigmoid activation function. The forget gate is trained so its output is close to 0 when part of the input is deemed irrelevant and close to one when the part of the input is deemed relevant. This vector is then pointwise multiplied with the previous cell state vector, which means that when a part of the cell state is determined to be not useful, it will be multiplied by a number close to 0, which will make it have less influence on the subsequent steps.

The next part is the input gate which determines which parts of the input should be added to the cell state (networks memory) given the previous hidden state and new input data. But before we get to the input gate, there is another network that is *tanh* (Hyperbolic Tangent)

activated (in Fig. 2.1 next to the input gate in blue) that needs to be explained first. The job of this network is to learn how to combine the previous hidden state and the new input data to create a new memory update vector. This vector contains information from the input data given the context from the previous hidden state and tells the cell how much to update each element of the cell state with new information. *Tanh* is used here to make values lie between -1 and 1 (the negatives allow the cell to reduce the impact of elements in the cell state). Once the new memory vector is created, the sigmoid-activated input gate network acts as a filter, deciding which parts of the new memory vector are worth remembering. Since the input gate is sigmoid activated, it is on the range 0 to 1, which is then pointwise multiplied with the new memory vector. If the output is near 0, then the magnitude of that element of the new information vector will be close to zero. This combined vector is then added to the cell state, which updates the long-term memory of the network.

Finally, the output gate is the last step in the LSTM cell and acts as a filter, deciding what is included and not included in the new hidden state. The output gate is a sigmoid-activated network that uses the previous hidden state and the new input data to create a filter. This filter is applied to the newly updated cell state to create the new hidden state. The previous cell state is first passed through pointwise hyperbolic tangent to force values into the interval -1 to 1, then is pointwise multiplied with the output of the output gate, creating the new hidden state that will be fed into the next step of the network.

2.2 Capturing Network Data

Network data is captured in one of two formats. One is packet-based, where all information about packets that flow through a networking device, including payload, is captured and stored usually in a PCAP file. Packet-level capture usually contains more information than flow-based. In addition, flow-based attributes can usually be extracted from a packet-based capture. For this reason, this thesis will focus on packet-level data.

The other format is flow-based capture which is a more condensed format that focuses on connections instead of individual packets. It is an aggregate of all the packets that flow from one computer to another. The flow-based format usually captures the source IP address, source port, destination IP address, destination port, and transport protocol at minimum. From there, it usually includes many extracted features about flows, such as connection time and the number of packets. One thing that is not included in flow-based capture is the payload or any information about specific packets.

2.3 Intrusion Detection Systems

An IDS is an application that monitors a network or device and searches for malicious activity. Most IDSs simply report suspicious activity so a member of a security team can investigate, but some are also active and respond by blocking or removing threats. There are two main types of IDSs, network-based intrusion detection systems (NIDS) and host-based intrusion detection systems (HIDS).

HIDS is a system that is installed on a specific device and its only job is to monitor the security of that device. It does this by analyzing system objects, logs, processes, regions of memory, and other parts of the host it is installed on [2]. Since it is a host-based system it can see information that a NIDS cannot since TLS encrypts application layer information when packets are in transit. This would be the type of IDS that would detect malicious URLs such as the ones mentioned in Chapter 4.

NIDS is different that HIDS since it analyses all incoming and outgoing network traffic that passes through a specific point in a larger network structure [21]. This means it monitors the network traffic of multiple hosts or devices on a network. It looks for specific patterns in the network packets or packet streams that could suggest malicious intent. However, since most data is encrypted in transit by transportation layer security (TLS) NIDS only have access to Ethernet, IP, UDP, and TCP header information. This is the type of IDS that is referenced in Chapter 6.

Chapter 3

Related Work

To my knowledge, anonymizing a packet-level network dataset using deep learning has never been attempted. However, what has been attempted is network dataset creation through deep learning and other methods. So, in this chapter, non-deep learning approaches to network dataset anonymization and network dataset creation techniques will be presented.

3.1 Anonymization Techniques

3.1.1 The Devil and Packet Trace Anonymization

In this paper, Pang et al. discuss a non-machine learning approach to packet trace anonymization that uses user-defined policies. They aim to strike a balance between the user's anonymization goals and retaining enough information to not diminish the research value of the traces [19]. They create a framework called **tcpmkpub** [11] which is customizable so the user can make explicit rules for each header field. The framework allows for custom transformation functions to be created and applied to a header field, or they can just be zeroed out. This is a simple way of anonymizing packet-level datasets. However, it suffers from the fact that the anonymization policies are human-made. This gives more control over what is anonymized but will probably end up removing attributes that could be important to a deep learning model.

The deep learning anonymization approach I present in this thesis moves away from human interference and allows a model, during training, to decide for itself what it thinks are the most important characteristics of a dataset to preserve. This allows for less control over what is anonymized. But, because a deep learning model anonymized the data, it should keep the characteristics that would be important to another deep learning model that is trained on the synthetic dataset it created. This leads to more information retention in the synthetic dataset in terms of what a deep learning model would need.

3.1.2 Mimic Learning to Generate a Shareable Network Intrusion Detection Model

This paper proposes using mimic learning [5] to get around the problem of sharing intrusion detection models that were trained on private network data that cannot be released publicly due to privacy concerns [22]. Mimic learning is a form of transfer learning [29] that involves a teacher and a student model. The teacher learns something and then, by some means, transfers its knowledge to the student. In this case, the teacher model is selected by training several classification models on the original private data and selecting the one with the best performance. The teacher model is then used to label publicly available data, which the student model will then be trained on. The goal of the teacher-student training process is to transfer the knowledge that was extracted from the original private data by the teacher to the student through the publicly available data the teacher labeled. This leaves them with a student model that indirectly learned the characteristics of the private data, which should alleviate privacy concerns when sharing the student model publicly. This indirect approach to sharing sensitive data publicly is similar to what I do in this thesis, except I focus on the data, and they focused on the model. The labeling of the publicly available data acts as a way to indirectly transfer knowledge. In a similar way, I use a character-level LSTM to learn the characteristics of a dataset and then recreate new synthetic data that has the same characteristics but is anonymized enough it could be shared publicly.

3.2 Network Traffic Generation

3.2.1 Network Traffic Data Generation with Generative Adversarial Networks

A paper by Cheng et al. discussed the use of Generative Adversarial Networks (GAN) [4] for the creation of IP packet-layer network traffic data [3]. The GAN has proven highly successful over the past few years with its ability to create highly realistic yet artificial images, text, audio, and video data [1]. Convolutional neural networks (CNN) GANs are used in this study. CNN GANs use CNN models for both the discriminator and generator. The researchers developed a new technique to encode network data specifically for use in CNN GANs. There are two steps to the encoding scheme: 1) converting packet byte values into subranges of sequential values, and 2) duplicating and mapping these converted values into a CNN input square matrix multiple times. Generating individual traffic types can lead to a success rate of up to 99%, whereas the generation of different traffic mixes produces a success rate of up to 88% [3]. This study shows excellent results using GANs to generate data, so I attempted a replication of their work that just focused on generating DNS packets.

The replication also showed good generation results sometimes in terms of correctly formed packets, but it was found that the generator model suffered from mode collapse, causing most of the generated packets to be the same. In the experimental results at maximum, the generator would have a success rate of 75-80%. This seems like a good result but upon visual inspection of the dataset, the variability was very low. Most of the packets were the exact same so although it created the packets correctly, almost all of that 75-80% was the exact same packet just generated over and over again. GANs have had great success in image generation due to the data being continuous but they continue to have problems generating discrete data; such as the data used in this thesis. Due to this, the generated data would not have been useful so it was decided not to use a GAN and instead use a character-level LSTM model that is often used in text generation. LSTMs do not have the same problems with discrete data CNN based GANs

have and their ability to process sequential data seems perfectly suited to network data as it is transmitted sequentially over communication wires.

3.2.2 Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization

This paper by Sharafaldin et al. does not use deep learning to create an intrusion detection dataset but rather captures it from a small network they created. They collect normal user behavior by using their B-profile system [23]. Then they use scripted attacks to attack the victim network and record all the network traffic [24]. The problem with this approach is the small network architecture doesn't accurately reflect what an IDS would see in the real world. It is just too small, which causes a lack of variability in IP addresses and other features. When an IDS is evaluated on the dataset, it will likely experience some overfitting and poor generalization performance when tested in a different networking environment. My research aims to overcome these shortcomings by learning the characteristics of datasets that came from large corporate networks. Then, creating an anonymized synthetic dataset that would have much higher variability in its features and be much more realistic as it came from a real networking environment. That being said, the dataset that is created from their research is still the best I could find and is the dataset I will use later in this thesis. More detailed information about this dataset can be found in Section 6.2.

Chapter 4

URL Dataset Generation and Anonymization

4.1 Introduction

[1]The problem of anonymizing and generating a TCP dataset is challenging. As stated in Section 1.3, a staged approach to this problem was taken. This approach started with a more manageable problem and built on the lessons learned at each step toward the most challenging problem of anonymizing a TCP dataset. That starting problem was decided to be the anonymization and generation of a URL dataset which is shown in this chapter. Starting with URLs was chosen because the domain part of a URL is part of the payload of a DNS query/response packet (which is the next step after URLs generation is confirmed possible). This makes URLs simpler than a complete DNS packet and thus an excellent place to start. URLs can also be benign or malicious, allowing them to be tested in a similar fashion to a malicious and benign TCP dataset. Making much of the tests developed in this chapter usable in the TCP section.

[1]A version of this chapter has been accepted for publication for publication [28]

4.2 Character-Level Long Short-Term Memory Models

Two character-level LSTM models were coded and trained in Pytorch, one to generate malicious URLs and one to generate benign URLs. The LSTM models consist of an embedding layer, an LSTM layer, a dropout layer, and a fully connected layer. The embedding layer embeds each of the integers in an encoded training sample into a vector that is the length of the character dictionary. The LSTM has two layers and a hidden size of 512. The dropout layer helps with overfitting and has a probability of 0.5. Finally, a fully connected layer is at the end of the model and takes the hidden size of 512 in, and has an output the same size as the character dictionary. This is because each of the output neurons contains the probability of each character in the character set being the next character in a sequence.

4.3 Dataset

In this experiment, the URL dataset we used was ISCX-URL2016 from the Canadian Institute for Cybersecurity [15]. It contains 35,377 benign URLs, 12,000 spam URLs, 11,566 malware URLs and 9,965 phishing URLs. The spam, malware, and phishing URLs are concatenated together to create the malicious URL dataset containing 33,531 samples.

4.3.1 Preprocessing

Before the dataset can be used in the character-level LSTM model, it must be preprocessed into something the model can use. The first step is to get all the possible characters in the list of URLs and make a dictionary that encodes the characters to integers and another that decodes the integers back to characters. The dictionary must also have the 0th entry as the special padding token. This padding token is used to pad the URLs in a batch since the URLs will vary in length, and LSTMs require a uniform length throughout a batch. It will also contain a special start token to indicate to the LSTM model the start of a sequence and a special end token that the LSTM model places at the end of a sequence to indicate the end of the generated sequence.

The dictionary is then used to encode the entire set of URLs into integer representations. An example of this encoding is shown in Fig. 4.1.

4.4 Training

The goal of training is for the character-level LSTM models to learn how to generate data with the same characteristics as the original URL data but not generate the same data. The first step is splitting the data into train and validation sets; 90%, and 10%, respectively. The training set was much larger since these models require a large amount of data to train. There is no test set because you can not test character-level LSTM models by comparing inputs to generated sequences. Their only input is the start token; then, they generate URLs one character at a time based on the probability of what the model thinks the next character should be. Next, the batches must be created.

Mini-batched training is standard practice in deep learning where instead of giving a neural network a single training sample at a time, it is given several. These training samples are put in a matrix which allows the losses to be calculated simultaneously rather than one at a time. The loss is then averaged and backpropagated through the network. If mini batches were not used, then backpropagation would have to be done after every sample's loss was calculated, which is very inefficient. Using mini batches overcomes this inefficiency and speeds up the training process considerably.

In a batch of size one, take a sample and make two arrays, one for input and one for target, each being of size one longer than the length of the sample (length of n+1). The input array has the start token placed in the first position of the array and then each character of the sample in the rest of the cells. The target array has the characters of the sample placed in the first n cells and the end token placed in the last cell (n+1). This creates an offset between the input and target arrays. This is because the LSTM is trying to predict the next character; therefore, if the input is the start token, then the next character (target) is the first character in the URL,

and onward [12].

The next part of the batching process deals with the URL samples all having different lengths. Since they have varying lengths to do batched training, you must find the lengths of all the samples, then sort them by longest to shortest, and then pad all the shorter sequences with the special padding character [25]. The special padding token is ignored in the loss function, and therefore, it does not contribute to the loss. An example of the batching and encoding process can be seen in Fig. 4.1.

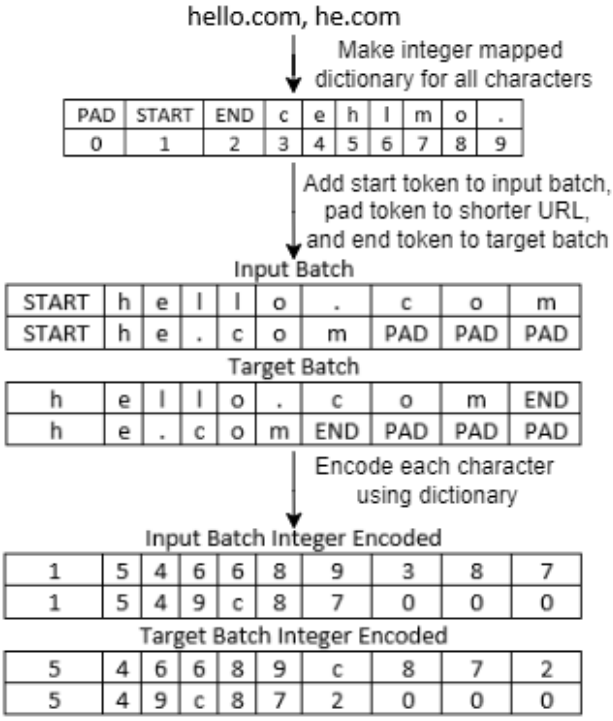

Figure 4.1: Example of encoding one batch that includes the domain part of two URLs hello.com and he.com.

The training function uses cross-entropy loss and Adams optimization [10] with a learning rate of 0.001, β_1 of 0.9, and β_2 of 0.999. It has a batch size of 128, and to ensure that we do not run into the exploding gradient problem with really long URLs, the gradient norm is clipped with a max norm of 5. Every epoch, a validation step is included, and if the validation loss doesn't improve after three epochs, then training was stopped. For information about the hardware environment see Appendix B.

4.5 Generating URLs

To evaluate the quality of the character-level LSTM models, two synthetic URL datasets had to be generated. One was made up of synthetic malicious URLs generated by the LSTM model trained on the real malicious URLs. The second was made up of synthetic benign URLs generated by the LSTM model trained on the real benign URLs. To generate the URLs, two custom functions were used, one called 'predict' and one called 'sample'.

The predict function takes in a character, and a hidden state. Then passes these to the model's forward function and obtains probabilities raw output and a new hidden state. The hidden state will be returned to the calling function in order to be passed to predict again if it needs another character. The raw output is given to the Softmax function, which outputs the probability that each character in the character dictionary should be the next character. This probability distribution is then passed to a random sample function that randomly selects one of the characters from the dictionary considering the Softmax probabilities. This is where the anonymization comes in, as the ransom sample function adds some selective randomness into the generated samples. This random sample function can be used to add more variation (anonymization) to the model by making it select more than one character and randomly selecting from those. This function then returns the chosen character and hidden state.

The 'sample' function calls the 'predict' function. The sample function takes in an LSTM model as input and outputs a fully formed URL string. It first passes the start token to the model and obtains a prediction for the next character, which is added to a list. It then continues to call the predict function passing in the most recently predicted character until the network predicts the special end token, which tells the algorithm the URL is fully formed, and it stops predicting and returns the URL string.

To illustrate the generation process recall the example of hello.com and he.com in Fig. 4.1. Now, if an LSTM was trained on these two URLs, it would definitely overfit, and we would most likely get the same URLs back when we generate samples. If we wanted it to generate a URL string, first, the 'sample' function shown in Fig. 4.2 would be called and it would

pass the start token to the trained LSTM's predict function, as can be seen in Fig. 4.3, which would put the start token through the LSTM and the output would be a probability distribution like the one shown in step 1 of Fig. 4.3 and a hidden state which is the short-term memory of the network. The distribution is the Softmax probability that each of the characters in the dictionary should be the next character in the URL string. In this case, the probability that "h" is next, is 1, because in the two training samples, "h" is the only character that ever comes after the start token, and the LSTM would have learned this. The random sample function would see this, select it as the next character, and "h" along with the hidden state would be passed back to the sample function. Now "h" is concatenated to the generated URL string and the first hidden state and "h" is passed back to the LSTM's predict function. Next, in step 2 of Fig. 4.3 it can be seen that "e" has a probability of 1 because "e" is the only thing that ever comes after "h" in the two training samples. So "e" is chosen as the next character, and it, along with the new hidden state, is passed back to the sample function. The sample function concatenates "e" to "h" and the URL is now "he". Next, "e" and the hidden state are then passed back to predict and the probability distribution in step 3 of Fig. 4.3 shows that "l" and "." both have a 0.5 probability this is because in hello.com "l" comes after "e" and in he.com "." comes after "e". This is where the random sample function comes in and will pick one or the other adding variability to the generated samples. In this example, it picks "l" as the next character. Now the URL string is "hel" so "l" is passed back to the predict function and the probability distribution may look like step 4 of Fig. 4.3. In this case "l" is predicted with a probability of 0.75 and "o" has a probability of 0.25. This case may arise because both "l" and "o" can possibly come after "l" in hello.com. The reason why "l" has a higher probability than "o" is because the LSTM's cell state (long-term memory) is allowing it to remember the letters that came before the "l" which were "h" and "e". It would know that there should probably be two "l's" before it predicts "o". This process continues until the LSTM predicts the end token should be next, at which time the sample function shown in Fig. 4.2 knows that the string is fully formed and to stop predicting and return it.

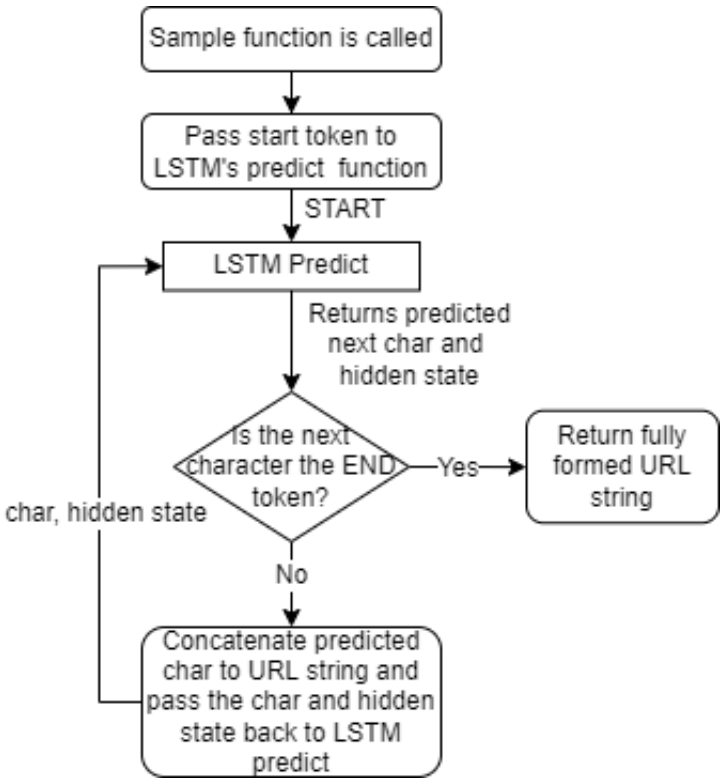

Figure 4.2: Sample function flow diagram.

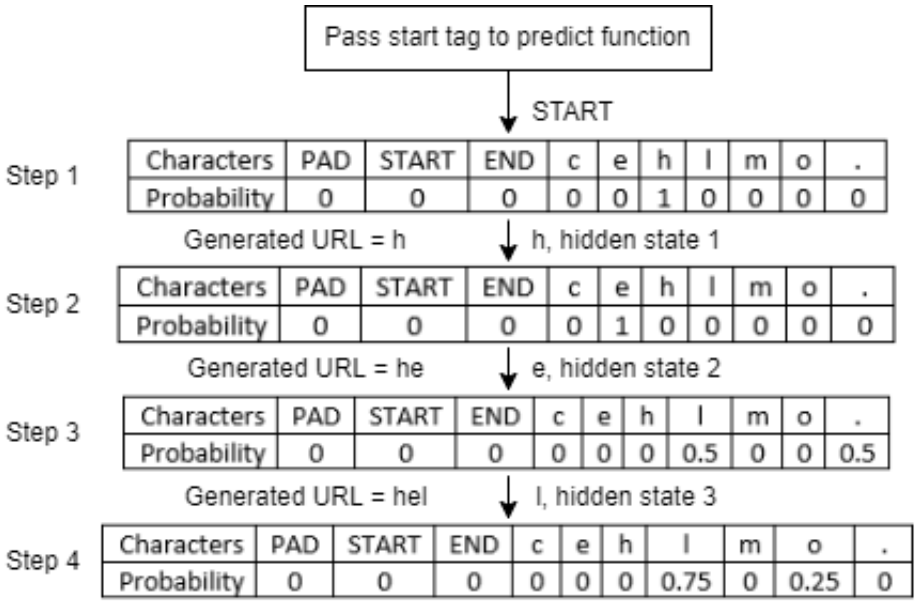

Figure 4.3: Prediction function example.

4.6 Evaluation

This research had two goals: to create data that was characteristically similar to the original in terms of classification and for that data to be different enough (anonymized) from the original data; therefore, it could be made publicly available[2]. To evaluate these requirements, two tests were performed. The first test attempted to classify the URLs into malicious and benign to satisfy the first requirement. The second test checked the similarity between the real and synthetic URLs to satisfy the anonymization constraint. For these tests to be performed, 50,000 malicious and 50,000 benign URL samples were generated. Then any URLs with illegal characters were removed, and any duplicates within each set were removed. This resulted in 48,890 synthetic benign samples and 40,541 synthetic malicious samples.

4.6.1 Classifier

The classifier is another LSTM model but modified to do binary classification. This test for characteristic preservation will involve giving the network a list of malicious URLs (class 1) and a list of benign URLs (class 0) and asking it to classify them into one of the two classes. For this test, two classifiers are trained, one on the real benign and malicious datasets and one on the synthetic benign and malicious datasets. These two classifiers are then tested on both the real and the synthetic data. So overall, there will be four parts to this test: train on real, test on real (TRTR), train on synthetic, test on synthetic (TSTS), train on real, test on synthetic (TRTS), and train on synthetic, test on real (TSTR). The performance of these classifiers is then compared, and if the classifiers have similar performance, we can say the real data's characteristics have been preserved in the synthetic dataset. Several metrics were used to compare the performance, including accuracy, recall, precision, specificity, and a confusion matrix. The classifier was trained with a learning rate of 0.001, binary cross-entropy with logits loss, a batch size of 128, and a two-layer LSTM with a hidden size of 512. It was allowed to

[2]The trained models can not be released as they would contain too much information about the real dataset.

go for a maximum of 50 epochs, but early stopping was set to 3; therefore, training will stop if the validation loss doesn't improve. The data split used was 75% train, 10% validation, and 15% test when doing TRTR and TSTS. When TRTS is performed, the entire synthetic dataset is used to test it, and since the classifier has not seen any of this data during training, there will be no leakage. The same is true for TSTR, where the entire real dataset will be used for testing since it was not used for training.

4.6.2 Levenshtein Similarity Test

The Levenshtein ratio was used to evaluate how similar a synthetic URL is to a URL in the real dataset. Using Levenshtein distance [17], one can measure the difference between two sequences. The Levenshtein distance between strings represents the number of single-character edits (insertions, deletions, and substitutions) necessary to convert one string into another. The Levenshtein ratio uses the Levenshtein distance to calculate the percent similarity between two strings or, in this case, two URLs. For example, two strings are exactly the same if they have a Levenshtein ratio of 100%. Using this ratio, every synthetic malicious URL is compared to every real malicious URL. Every synthetic benign URL is compared to every real benign URL, and the highest similarity ratio is recorded. These are then plotted on two histograms to visually represent the two similarity distributions.

If this method was being used in an application outside of this thesis, the user would decide on a similarity tolerance level. Then, based on this parameter, URLs above a certain similarity ratio can be removed from the synthetic dataset. This allows different levels of anonymization to be met.

4.7 Results

4.7.1 Malicious URL LSTM model

The first model that was assessed was the malicious URL LSTM. The training took place over 18 epochs and the model converged to a training loss of about 0.6 and a validation loss of around 0.8. A visual representation of this can be found in Fig. 4.4.

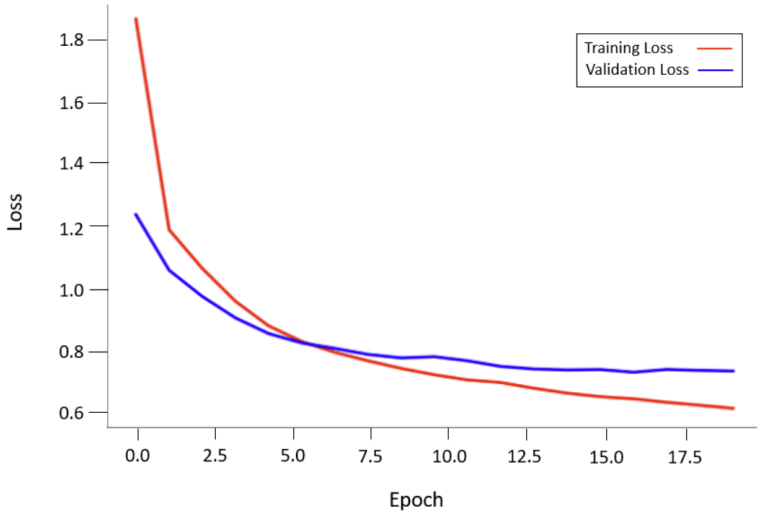

Figure 4.4: Malicious URL LSTM model training graph.

The Levenshtein ratio histogram shown in Fig. 4.5 shows how similar a synthetic malicious URL is to URLs in the real dataset. Looking at the histogram, it can be seen that most of the data points are between 0.5-1.0. The average value for this histogram is 0.78913. This indicates that, on average, a synthetic data point is 78.913% similar to a data point in the real dataset. The maximum value that the histogram got was 1.0. This indicates that when the model created data, it created some copies of the real data points. The minimum value the histogram got was 0.15.

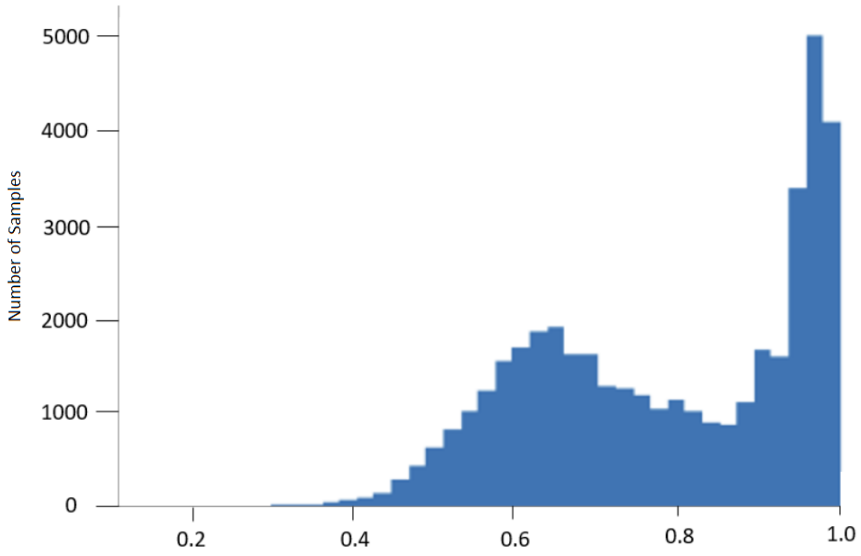

Figure 4.5: Similarity of generated malicious URLs to real URLs used in training.

4.7.2 Synthetic Malicious URL Samples

Below are some examples of synthetic malicious URLs that were created by the LSTM. As you can see, the URLs have proper top-level domain suffixes and are formatted correctly. Some include HTTP, and some are HTTPS. Some include the www. and some do not, but all this is good as it shows the model has good diversity and that it is learning the URL syntax. Examples of synthetic, malicious URLs:

- `http://amazon.co.uk/s/ref=sr_nr_n_2_2/202-2513040-1206232?ie=UTF8&rh=n:195522`
- `http://9779.info/%E8%A1%8D%E7%BA%B8%E8/`
- `http://www.accontamparoeb.com/wp-content/plugins/content/tmpl/ch/modules/mod_memage.css`
- `http://www.freatusa.com/ilap/images/nmh851/aol?products_readedpartnerId=2&nid=05`
- `https://spreadsheets.google.com/spreadsheet/viewform?formkey=dGJViHLWECjGUJMD3G2QC3==/`

4.7.3 Benign URL LSTM model

The benign URL LSTM was the next model to be evaluated. The training period was 27 epochs. In training, the training loss of the model converged to a value near 0.8 and its validation loss converged to a value of about 1. This is illustrated in Fig. 4.6.

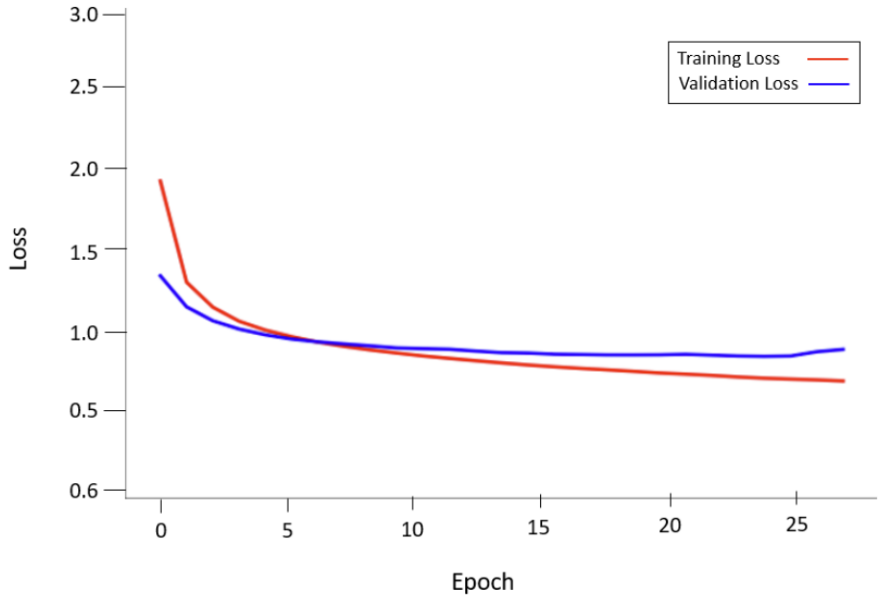

Figure 4.6: Benign URL LSTM Training Graph

According to the histogram shown in Fig. 4.7, the majority is around 60% similar. This histogram has an average value of 0.6719. As a result, on average, a synthetic data point has a 67.19% similarity to a real data point. A maximum value of 1.0 was obtained in the histogram. The minimum value the histogram got was 0.16.

4.7.4 Synthetic Benign URL Samples

Below are some examples of synthetic benign URLs that were created by the LSTM. Again the URLs have various schemes, subdomains, second-level domains, and top-level domains and are formatted correctly. Often, the domains in the synthetic samples are real; however, the subdirectory part of the URLs often leads nowhere, which is the anonymization part of the algorithm working.

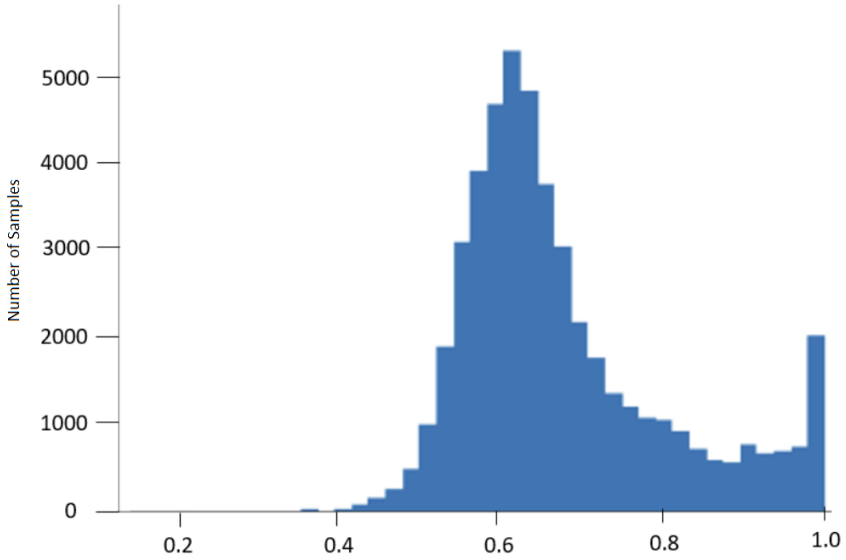

Figure 4.7: Similarity of generated benign URLs to real URLs used in training.

Examples of synthetic, benign URLs:

- `https://medium.com/dan-sanchez/duke-your-sony-couple-to-drab-temple-fakes-happening-with-mars-possibless`
- `https://www.gov.uk/government/organisations/review-in-detective-second-pental-illegal-goods`
- `http://kickass.to/bundles-player-2-killing-kongs-season-6-week-192s-song-crash/`
- `https://wordpress.org/showcase/electronics-dad-100-onlines-mailcategory/%5%35`
- `http://google.com/big/pep-watch-the-way-world-constructions-for-a-san/`

4.7.5 Classifier Trained on Real Data

The classifier trained on real data was the next model to be evaluated. There were 25 epochs during the training period for this model. The model showed a very low training loss, converging to around 0.005. Additionally, the validation loss was extremely low, converging to about 0.01. This can be seen in Fig. 4.8. After the model was trained, the model was tested twice,

once using the real test set and once using the synthetic dataset. The results of the test sets are shown in Table 4.1.

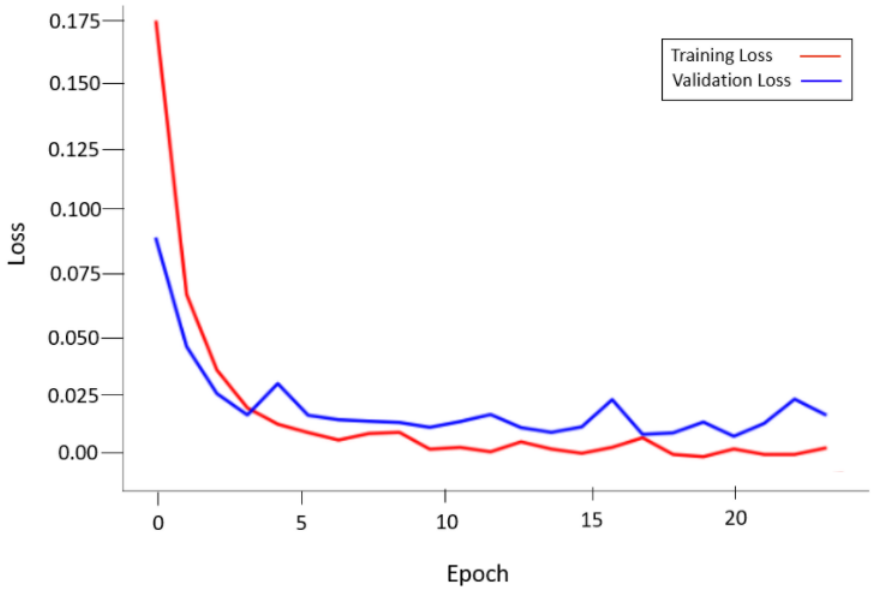

Figure 4.8: Classifier trained on real data training graph.

Metrics	TRTR Values	TRTS Values
Accuracy	0.9966	0.9873
Recall	0.9967	0.9952
Precision	0.9951	0.9772
Specificity	0.9965	0.9807

Table 4.1: Classifier Trained on Real Data Test Results

4.7.6 Classifier Trained on Synthetic Data

Next, the synthetic data classifier was evaluated. There were 10 epochs during the training period for this model. During training, the model showed a low loss, converging to about 0.1. Furthermore, the validation loss was also relatively low, converging to about 0.15. This can be seen in Fig. 4.9. After the model was trained, the model was tested twice, once using the synthetic test set and once using the real dataset. The results of the test sets are shown in Table 4.2.

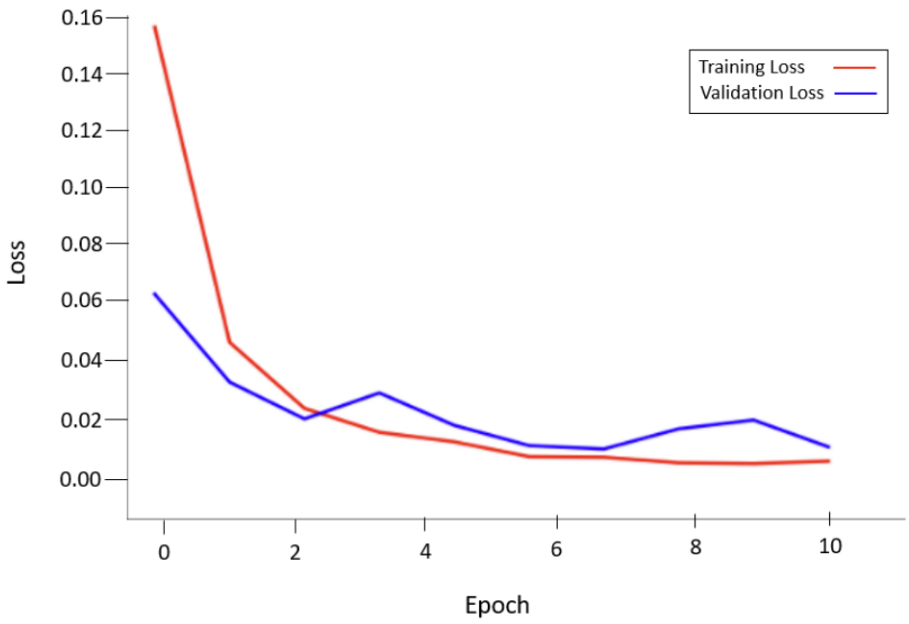

Figure 4.9: Classifier trained on synthetic data training graph.

Metrics	TSTS Values	TSTR Values
Accuracy	0.9957	0.9906
Recall	0.9948	0.9857
Precision	0.9958	0.9913
Specificity	0.9966	0.9940

Table 4.2: Classifier Trained on Synthetic Data Test Results

4.7.7 Confusion Matrices

The confusion matrix provides a visual representation of how a classification algorithm performs on its test set and what types of errors it is making. The confusion matrix provides count values for the number of correct and incorrect predictions. Ideally, numbers should appear on the diagonal and zeros elsewhere. Numbers outside the diagonal indicate that the model's performance is not perfect.

Starting with the classifier trained and tested on the real dataset in Fig. 4.10, it can be seen that the model does well overall. Looking at the diagonal on the matrix, there are only

a few mislabeled data (30 out of 8,960). This indicates that the classifier can identify which real URL links are benign or malicious with high accuracy. Next, in Fig. 4.11, you can see how the classifier trained on the real data performs when classifying the synthetic dataset. It only misclassified 1136 out of 89,344 synthetic samples showing it can accurately distinguish between the malicious and benign samples even when it was not trained on the same dataset.

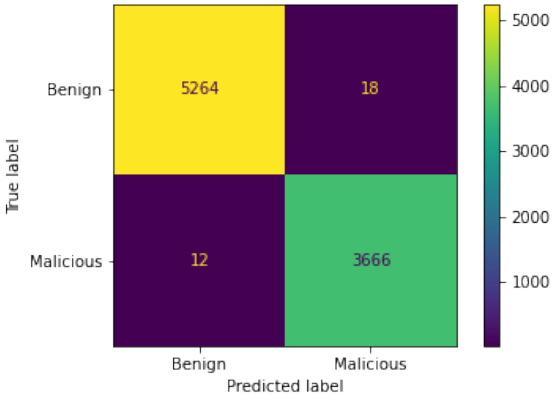

Figure 4.10: TRTR classifier confusion matrix.

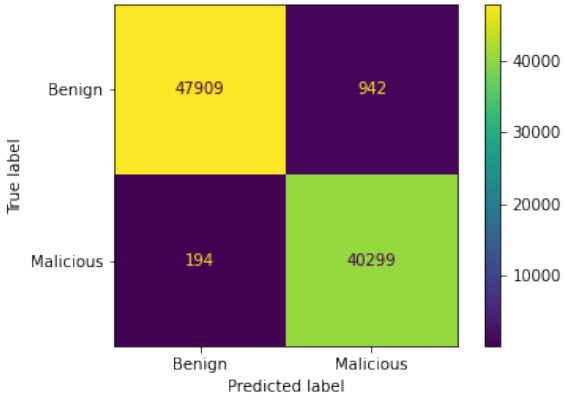

Figure 4.11: TRTS classifier confusion matrix.

The confusion matrix of the classifier trained and tested on the synthetic dataset shown in Fig. 4.12 can be seen to be very similar to Fig. 4.10, only mislabeling 56 out of 13,312 samples. As with the confusion matrix of the real dataset, only a few data points have been mislabeled in this dataset. Finally, in Fig. 4.13, you can see how the classifier trained on the synthetic data performs when classifying the real data. It only misclassified 562 out of 59,904 real samples showing its ability to classify the malicious and benign samples even when the dataset is different from the one it was trained on.

The fact that all confusion matrices have a very small amount of false positives and false negatives shows that the performance of the two classifiers is similar, suggesting that the synthetic dataset accurately captured the characteristics of the real dataset.

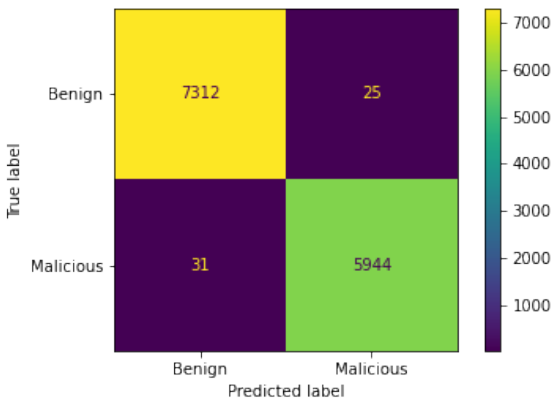

Figure 4.12: TSTS classifier confusion matrix.

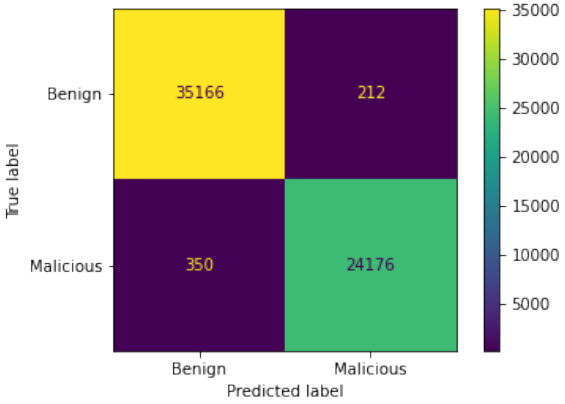

Figure 4.13: TSTR classifier confusion matrix.

4.8 Discussion

The results from the TRTR and TSTS classification tests show that the synthetic data classifier only slightly underperformed the real data classifier; however, with having accuracy, precision, recall, and specificity above 99%, it's easy to conclude their performance is similar and excellent. However, these metrics alone can't prove that characteristic preservation was achieved. The TRTR is mainly to show the best performance that could be achieved using only real data, and the rest of the tests should strive to be as close to this as possible. The results of TSTS being very close to this TRTR is very good, but the data could be in a format that is only usable by character-level LSTM models, or there could be mode collapse in the model. This is where TRTS and TSTR come in.

TRTS is important because it shows that when a classifier only sees real data during training, it is still able to classify the synthetic data with very similar accuracy. This means that the syntax used in the real URLs was captured by the LSTM that generated the synthetic data, and the data it created should be usable by any model in the same way the original data was used. This is shown in Table 4.1, where All metrics are only different by approximately 0.01-0.02, which is a great result. However, this test still wouldn't pick up mode collapse in the model, but it is still a good result that shows some characteristic preservation.

Finally, and most importantly is TSTR, as it will be able to detect mode collapse in the model, and it represents the main use case where a company could outsource cybersecurity development using the synthetic dataset to alleviate privacy concerns. In Table 4.2, you can see that again, the TSTR is only off from TRTR by about 0.01 in all metrics, which is an excellent result. This result shows that there was no mode collapse in the model since there is enough diversity in the synthetic training set that the model was able to classify almost all of the real data correctly. This means that a company could use the synthetic dataset to develop a machine learning model and be confident that when it is deployed back at the company that hired them, on real network data, the results should be almost the same. TSTR combined with TSTS and TSTR and compared to TRTR show that the characteristics of what makes a malicious URL malicious and a benign URL benign were preserved in the generated dataset.

One can visually verify the characteristic preservation by looking at the examples in Sections 4.7.2 and 4.7.4. You may also notice that the last example in Sections 4.7.2 and 4.7.4 are both google websites however they have different subdomains. The classification model can correctly classify these but defacement and typosquatting URLs were not used in the training data, so it will be left to future work to see if the algorithm can deal with these situations.

The Levenstein ratio tests showed a mean of 67% similarity for the benign URLs and a mean of 79% similarity for the malicious URLs. 67% for the benign URLs shows a sufficient level of anonymization, with some still being exact copies of the real ones; however, these can be discarded from the final set. The malicious URLs had more similarity with a mean of 79%, but this still should be sufficient anonymization. However, the malicious dataset had a significant amount more samples with 100% similarity. I think this can be attributed to the datasets used to train the character-level LSTM models. The benign LSTM model's dataset had around half a million unique URLs initially, and then after duplicates and domain-only URLs were removed, it was down to a little over 35,000. When manually inspecting the dataset, it seemed to have more variance than the malicious dataset. On the other hand, the malicious dataset had many URLs with the same domain (e.g. www.google.ca/) but slightly

different queries or paths added to the domain (e.g. https://www.google.ca/search?q=hello, https://www.google.ca/search?q=goodbye). I think this caused some overfitting and caused the LSTM model to generate many URLs with the same domain but different paths attached, which increased the mean similarity. This problem could be solved by finding a dataset with more variance in the malicious URLs. Despite this, the character-level LSTM model successfully generated an anonymized synthetic dataset that was characteristically similar to the original.

Chapter 5

DNS Packet Generation and Anonymization

5.1 Introduction

The next logical step on the way to generating and anonymizing a full network dataset after successfully generating and anonymizing a URL dataset would be generating domain name service (DNS) packets. DNS packets themselves contain the domain part of a URL as their payload but have added header information that increases the complexity of the generation process. This added header information brings us closer to the goal of generating and anonymizing a TCP dataset while still being slightly less complicated. DNS uses user datagram protocol (UDP) as its transport protocol which involves less header information making it simpler than TCP and an excellent second step. However, finding a suitable dataset was a challenge, as DNS, by nature, contains a significant amount of semi-public information. By semi-public I mean that since DNS is unencrypted so it can be viewed by a skilled third party. However, it is not public as people do not go out of their way to share their DNS data. This is because a dataset consisting of DNS packets would contain every website a user tries to visit. Fortunately, generating a simple DNS dataset is easy; all I had to do was record my network traffic. This is a secondary

reason why DNS was chosen as the next step since creating a dataset for training could be done quickly.

5.2 Dataset

The first thing needed for the generation process is a good quality DNS dataset consisting of a large quantity of DNS packets that have a lot of variability in the domains they query. This sample data also needs to be a decent approximation of the population, which in our case is a corporation. Wireshark was used on my home network and due to the current covid-19 pandemic, most people are working from home so this should be a reasonable approximation of the population. Even though it's not ideal, it is the best I was able to do with the time and resources at hand. To accurately represent a typical work week, data was collected over a seven-day period starting Monday at 12 am and ending Sunday at 11:59 pm. This data was then filtered to remove all protocols except DNS queries. After packet capturing was finished there were 75,460 DNS packets in the dataset. The packets were then exported as a JSON file in byte stream format. In Fig. 5.1 you can see an example of an exported DNS packet in byte string format. After they are in byte string format, the preprocessing steps are the same as in Section 4.3.1 except instead of the dictionary having many characters it is limited to the start, end, and padding tokens and the hexadecimal digits.

0000000000022cdb07c71bd60800/4500003c52bb000080116691c0a80013c0a8
0001/f8b2003500283fc1/c2f501000001000000000000056c6f67696e046c69766
503636f6d0000010001

Figure 5.1: Exported DNS packet byte string.

Each packet has four parts as shown in Fig. 5.1 and they have been separated by red slashes for clarity. The first part is the ethernet header which was discarded. It was decided to discard it so as to not add unnecessary complexity to the problem at this stage. In a hypothetical case where there was a problem with generating the DNS packets, I wanted there to be as few variables as possible that could be causing the problem so that finding it would be easier. In

the next chapter, the ethernet header is included, so it was not deemed necessary to include it in this chapter. The second, third, and fourth parts are IP header, UDP header, and DNS query. Fig. 5.2 shows each part of the packet in more detail.

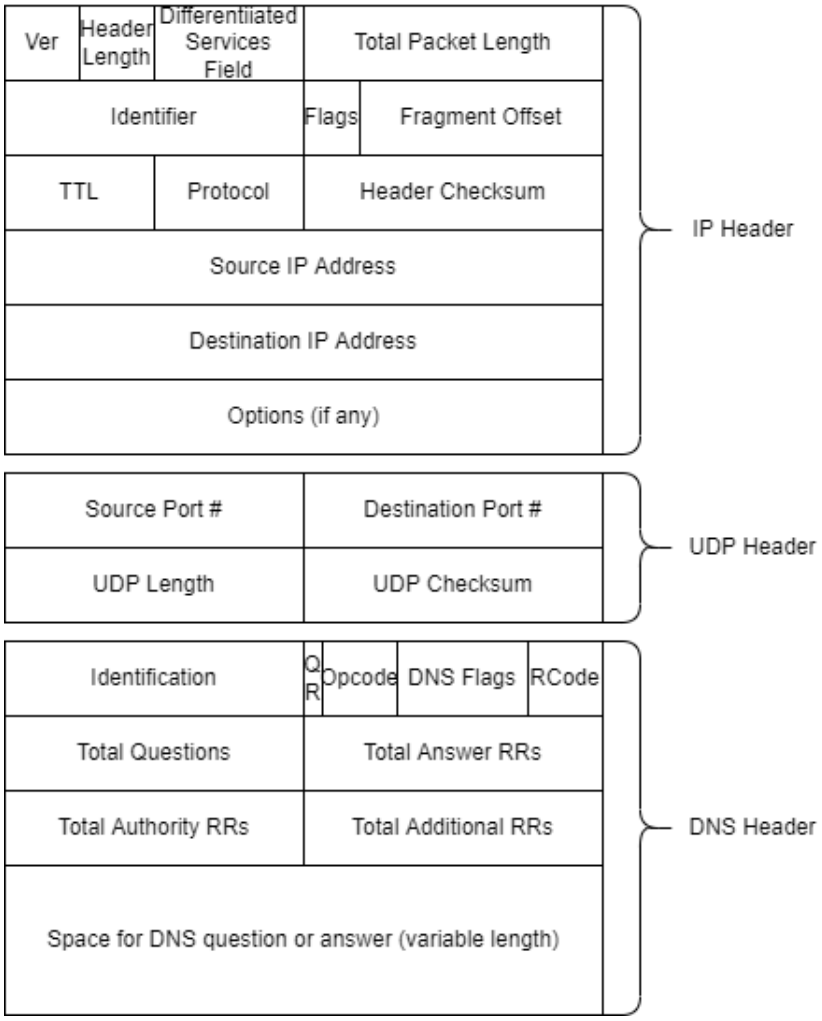

Figure 5.2: DNS Packet Structure.

Fig. 5.3 shows the DNS string put into human-readable format. The first section is the IP header, the second section is the UDP header and the third is the DNS Query. Under the Question section, the domain ”login.live.com” can be seen, which is the address being looked up.

IP Version: 4
Header Length: 5
Differentiated Services Field: 0x0
Total Length: 60
Identification: 0x52bb
Flags: 0x0
Fragment Offset: 0x0
Time to live: 128
Protocol: 17
Header checksum: 0x6691
Source IP: 192.168.0.19
Destination IP: 192.168.0.1

Source Port: 63666
Destination Port: 53
Length: 40
Checksum: 0x3fc1

Query ID: 49909 opcode: QUERY, flags: rd, rcode: NOERROR
Question Count: 1, Answer Count: 0
Authority Count: 0, Additional Record Count: 0
Name: login.live.com, Type: A, Class: IN

Figure 5.3: DNS example in human readable format.

5.3 Training

The training steps are the same as in Section 4.4. To start, the training data is split into 90% train and 10% validation. Then the training and validation batches are created. The training function uses cross-entropy loss, and Adams optimization [10] with a learning rate of 0.001, beta one of 0.9, and beta two of 0.999. It has a batch size of 128, and to ensure that we do not run into the exploding gradient problem with really long DNS packets, the gradient norm is clipped with a max norm of 5. A validation step was included in every epoch, and if the validation loss did not improve after three epochs, then training was stopped. For information about the hardware environment see Appendix B.

5.4 Generating DNS Packets

The steps to generate DNS packets are the same as in Section 4.5, except 5,000 samples were generated for each model, and any malformed packets were removed from the synthetic data set.

5.5 Evaluation

5.5.1 Characteristic Preservation Tests

Unfortunately, unlike the URL dataset, there is no malicious half to this dataset (we did not consider malicious DNS attacks), so no classification test was possible. Instead, two tests were performed on a generated dataset of 5,000 DNS packets. The first test checked that the packet structure was preserved in the synthetic dataset, and the second test checked for mode collapse in the model. The first test was performed using a DNS parsing tool that parsed the generated byte string into a fully formed DNS packet and checked for any errors in packet format. The parsing tool is written in python and throws an error when there are byte errors. This was exploited by using a try-except block where the parsing tool attempts to parse the byte string, and if successful, it adds one to a counter. If it is unsuccessful, it hits the except block, and nothing is added to the counter. This is put in a loop, and each generated sample was tested. When each sample had been checked, the percent correctly generated was calculated by dividing the successfully generated count by the total of 5,000 samples.

The second test used uniform manifold approximation and projection (UMAP) to dimensionally reduce the data from its 27 components which can be seen in Fig. 5.3 to 2 components so it can be plotted on an x and y-axis for visualization. The transformation from high to low dimensionality is learnable, allowing us to train it on real data only, then apply the transform to an unseen test set of the real data as well as the synthetic data. The results of the transformation are then plotted in an overlapping fashion. This overlapping graph allows us to analyze the

modes of the data to ensure characteristic preservation has been achieved. More information about UMAP analysis and its hyperparameters can be found in Section 6.7.3.

5.5.2 Levenshtein Similarity Test

The anonymization test will be performed the same way it was for the URL dataset using the Levenshtein similarity metric. In this test, each generated DNS sample will be checked against each real DNS sample to see how similar they are. The highest similarity percentage for each sample will be kept and plotted on a histogram for visualization, with 100 percent being the same and 0 being completely different.

5.6 Results

To maximize the utility of doing DNS generation as a second step towards TCP generation, the evaluation was performed on 3 different models in order to compare the effect of different numbers of neurons in the LSTM layers. The number of neurons tested is 256, 512, and 1024.

5.6.1 LSTM With 256 Neurons

Training lasted 61 epochs and ended with a lowest validation loss of 0.3446 at epoch 58. It had an epoch time of about 46 seconds each, plus 2 minutes and 12 seconds per epoch to shuffle and re-batch the training data for a total training time of about 3 hours 1 minute. In Fig. 5.4 a graph of this model's training can be seen. When generating the 5,000 synthetic samples, 889 had incorrect DNS packet structure and were discarded. This resulted in a successful generation rate of 82%.

In terms of anonymization, the Levenshtein ratios histogram in Fig. 5.5 shows a mean value of 90.11% similarity, a maximum value of 96.35% similarity, and a minimum value of 57.75% similarity.

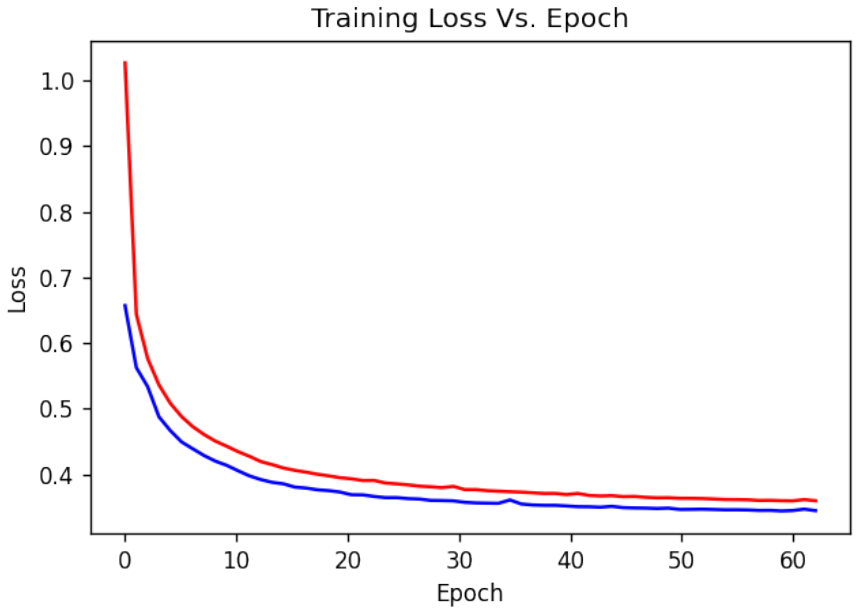

Figure 5.4: LSTM model with 256 neurons training loss (red) and validation loss (blue) vs epoch.

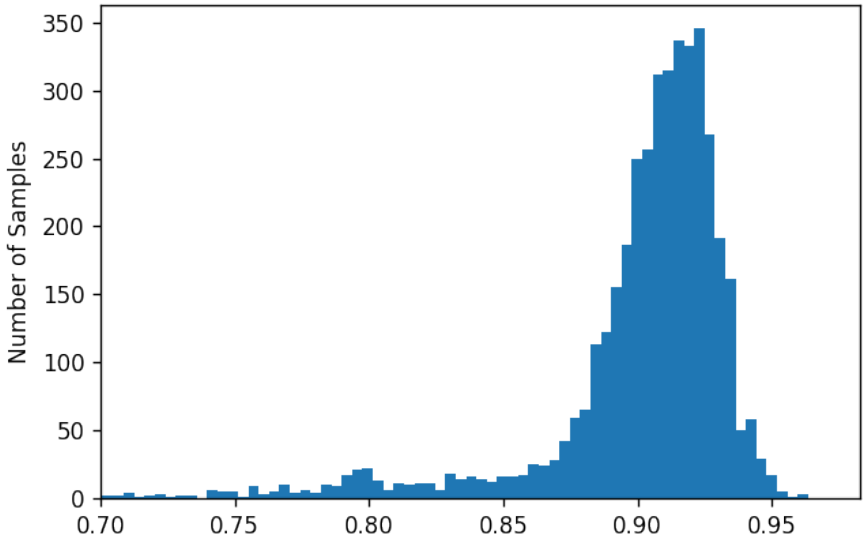

Figure 5.5: Levenshtein ratios histogram for LSTM with 256 neurons.

5.6.2 LSTM With 512 Neurons

Training lasted 77 epochs and ended with a lowest validation loss of 0.3016 at epoch 74. It had an epoch time of about 1 minute 22 seconds each, plus 2 minutes and 12 seconds per epoch to shuffle and re-batch the training data for a total training time of about 4 hours 35 minutes. In Fig. 5.6 a graph of this model's training can be seen. When generating the 5000 synthetic samples, 585 had incorrect DNS packet structure and were discarded. This resulted in a successful generation rate of 88%.

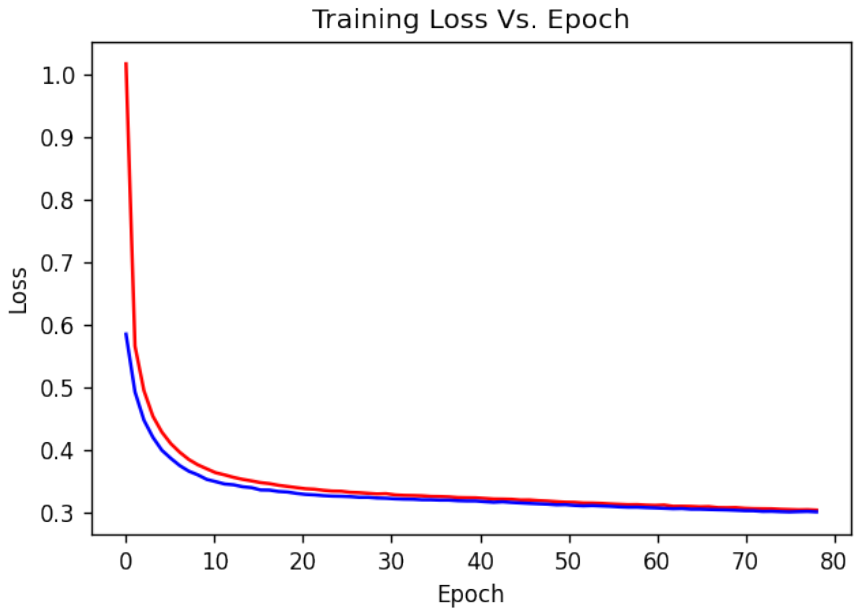

Figure 5.6: LSTM model with 512 neurons training loss (red) and validation loss (blue) vs epoch.

In terms of anonymization, the Levenshtein ratios histogram in Fig. 5.7 shows a mean value of 90.74% similarity, a maximum value of 96.64% similarity, and a minimum value of 60.43% similarity.

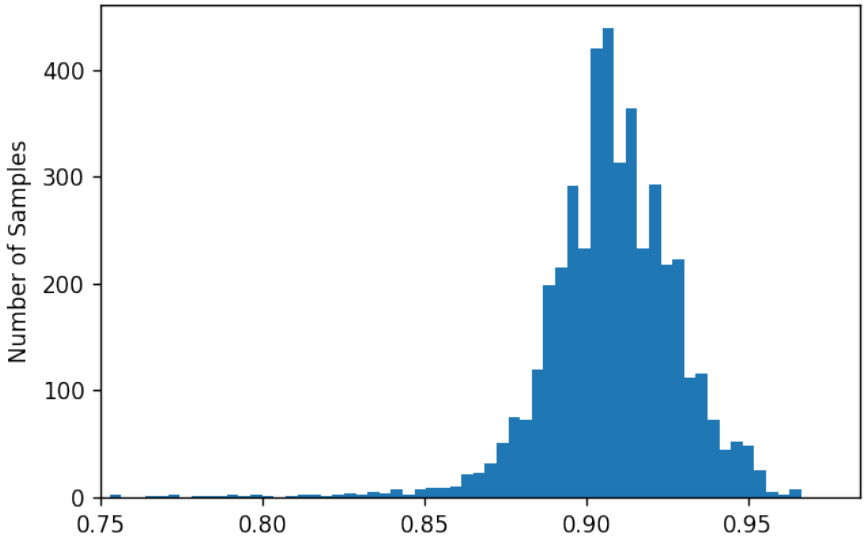

Figure 5.7: Levenshtein ratios histogram for LSTM with 512 neurons.

5.6.3 LSTM With 1024 Neurons

Training lasted 83 epochs and ended with a lowest validation loss of 0.2453 at epoch 80. It had an epoch time of about 3 minutes 6 seconds each, plus 2 minutes and 12 seconds per epoch to shuffle and re-batch the training data for a total training time of about 7 hours 20 minutes. In Fig. 5.8 a graph of this model's training can be seen. When generating the 5000 synthetic samples, 200 had incorrect DNS packet structure and were discarded. This resulted in a successful generation rate of 96%.

In terms of anonymization, the Levenshtein ratios histogram in Fig. 5.9 shows a mean value of 91.38% similarity, a maximum value of 97.50% similarity, and a minimum value of 64.54% similarity.

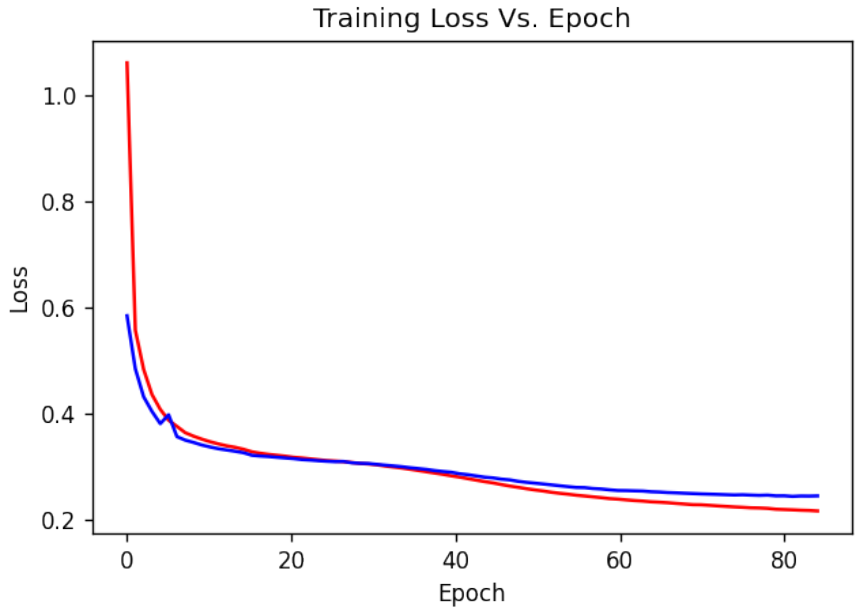

Figure 5.8: LSTM model with 1024 neurons training loss (red) and validation loss (blue) vs epoch.

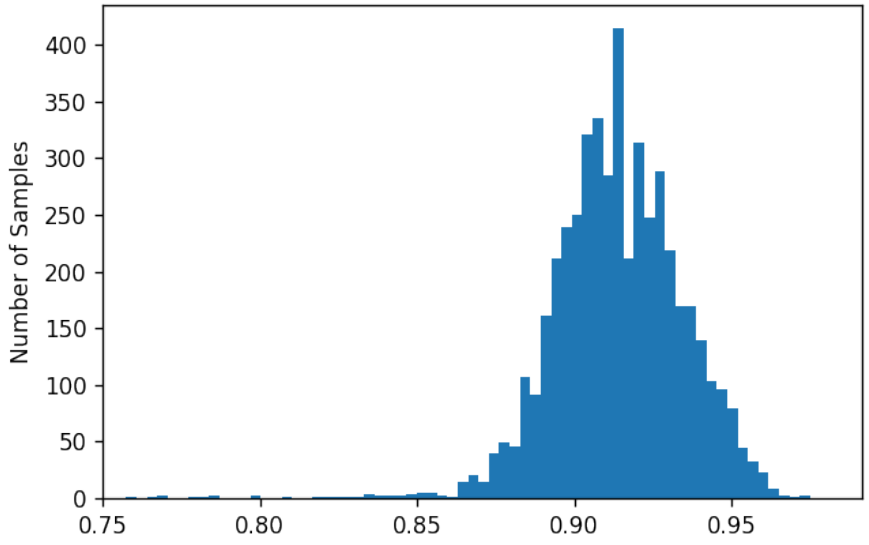

Figure 5.9: Levenshtein ratios histogram for LSTM with 1024 neurons.

5.6.4 UMAP

Here the real DNS samples were dissected into their packet attributes like in Fig. 5.3. Those attributes were then converted from their hexadecimal values into integers and then scaled by removing the mean and scaling them to unit variance. This process was also done for each of the 3 model's synthetic samples to create 3 out of sample test sets for the UMAP model. A UMAP model was then trained on 90 percent of the real dissected packet data, and 10 percent was left as an in-sample test set. The trained model was then tested on the real DNS data and the 3 out of sample synthetic datasets. The results of the real and synthetic data tests were plotted on the same graph for visualization. The test set size was not made larger because if it is too large, the points will overlap too much, and it is very hard to see the spread in the data points. 10% seemed reasonable as it would be about 7,500 real data points, and then the 5,000 synthetic samples would be plotted on top of the 7,500 real points. This allowed for a visualization where the points could be differentiated and also a good sample size. In Fig. 5.10, Fig. 5.11, and Fig. 5.12 you can see the results of the UMAP dimensionality reduction for the LSTM with 256, 512, and 1024 neurons respectively.

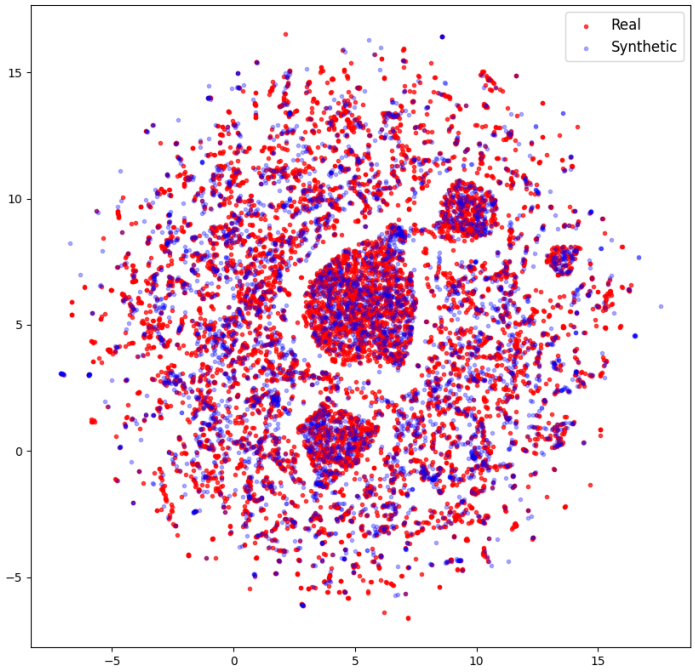

Figure 5.10: UMAP model trained on real DNS data and tested on synthetic DNS data generated by LSTM model with 256 neurons.

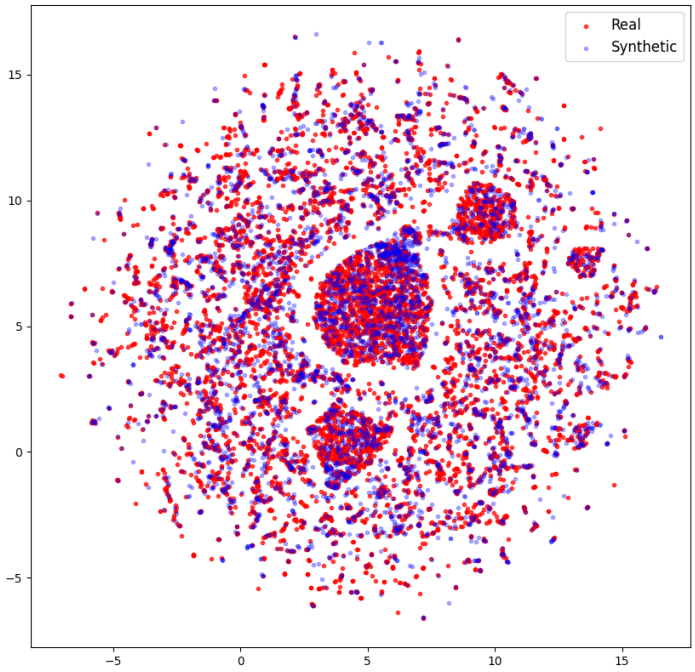

Figure 5.11: UMAP model trained on real DNS data and tested on synthetic DNS data generated by LSTM model with 512 neurons.

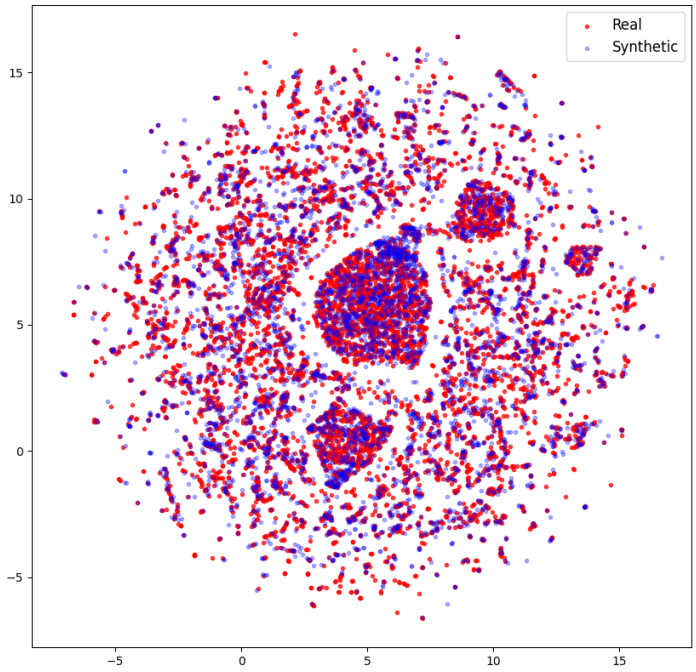

Figure 5.12: UMAP model trained on real DNS data and tested on synthetic DNS data generated by LSTM model with 1024 neurons.

5.7 Discussion

The three models ended up with generation success rates of 82% for the model with 256 neurons, 88% for the model with 512 neurons, and 96% for the model with 1024 neurons. This is a 6% increase from 256 to 512 neurons and an 8% increase from 512 to 1024 neurons. However, this also comes with a training time increase from 3h to 4.5 hours to 7 hours. Showing that while the extra neurons allow the LSTM model to learn the packet structure better, it also significantly increases training time.

In Figures 5.5, 5.7, and 5.9 the results of the Levenshtein similarity ratios can be seen in histogram format for the models with 256, 525 and 1024 neurons respectively. The first thing that can be seen is that there is a left skew to each of the plots, but when you go from 256 to 512 to 1024 neurons, that skew becomes less and less. The graphs for 512 and 1024 neurons in Figures 5.7 and 5.9 have their x-axis's limited to 0.75. This is because there are almost no data points after that, and it allows them to be on the same scale and for the bulk of the

graph centered around 90 to be clearer for analysis and comparison. Similarly, the graph for 256 neurons in Fig. 5.5 has its x-axis limited to 0.70 for the same reasons; however, I could not make it 0.75 like the others because it had a significant amount of points up to 0.70. The left skew in Fig. 5.5 is probably due to the LSTM not having the capacity to learn when to stop generating a sample and probably had some packets that were extremely long. This would make them very different from the real packets due to their extreme length causing them to have a lower similarity. This effect disappears more in in Fig. 5.7 and is almost completely gone in Fig. 5.9. This shows that increasing the number of neurons increases the network's capability to lean the packet structure with diminishing returns. For this reason, there was no need to go past 1024 neurons as it would most likely lead to overfitting and decreasing the anonymization capabilities of the network.

Besides the skew, the Levenshtein graphs are pretty similar, all having a mean of around 90% similarity. This is pretty high, so not a lot of anonymization was performed. However, this can be attributed to low variance in the dataset. All the data was collected from my computer due to the unavailability of a suitable dataset elsewhere, so things such as IP addresses will only have a few values. In this case, the IP addresses only have two values, one for my machine's IP and the other for my router's IP, as the packets are sent to and from my router. This causes overfitting on these parameters leading to almost no variability in the synthetic dataset. However, I believe that with a larger, more diverse dataset, the mean similarity would be much lower as the LSTM would not overfit on parameters such as IP.

The first thing that can be seen in the UMAP graphs in Figures 5.10, 5.11, and 5.12 is that there are three modes in the graphs. What these modes are is not important; the only thing that matters is that the synthetic (blue) data points also have these three modes. This shows that in all three cases, the LSTM models can generate a synthetic dataset that conserves the global structure and characteristics of the real data. However, the three models are able to conserve this structure at differing rates. The model with 256 neurons in Fig. 5.10 is definitely the worst, which can be seen if you compare it to the model with 512 neurons in Fig.5.11. The blue dots in

Fig. 5.10 are much more spread out, with less of them clustering inside the modes, which can be seen the best at the top of the center cluster when compared to the model with 512 neurons in Fig. 5.11. In Fig. 5.11 the model with 512 neurons strikes a good balance between conserving the global structure and modality while also having blue points that do not directly overlap with the red points meaning they are distinct from the real data points. This, incorporated with its 88% generation success rate, shows that it has done some anonymization of the DNS data while still preserving the characteristics of the real dataset without experiencing mode collapse. The model with 1024 neurons seen in Fig. 5.12 looks very similar to the model with 512 neurons seen in Fig. 5.11. The only difference is that the three modes may have slightly more data points clustered in there, and the outer points may overlap with the red a little more, but it is hard to tell.

The success rates, along with training time, adequate anonymization, and the UMAP graphs, show that either 512 or 1024 neurons would be an acceptable choice, but 256 neurons is too small. So it comes down to if a user is constrained by time and resources. If they are not, then the model with 1024 neurons generation success rate of 96% would be a good choice as long as it is acceptable to lose a little anonymization. If a user has less resources and time, then the model with 512 neurons would also work well. However, I would anticipate the model with 1024 neurons to show a bigger advantage on a dataset with more samples and more diverse attributes.

Chapter 6

TCP Generation and Anonymization

6.1 Introduction

After the successful generation and anonymization of both a URL and DNS dataset the next and final step was to generate and anonymize a packet-level TCP dataset. Finding a packet-level TCP dataset that included benign and malicious network traffic is extremely difficult which just reinforces my belief that this research is needed. All the datasets I found were either outdated and from the late 90s or early 2000s or only had TCP traffic flows. TCP traffic flows focus on extracted features from TCP packet datasets such as the number of packets or connection time. Using extracted TCP flow features is useful for some applications but, a lot of data that could be useful to a machine learning algorithm is lost in the process. This loss of information is primarily why it is done, as it acts as a form of anonymization since the actual packets themselves do not have to be released. In this section, I offer an alternative solution where an LSTM network is used to learn the characteristics of the packet-level dataset and recreate a new anonymized version of that dataset. This way the LSTM model itself will decide which characteristics are important to the structure of packets in this specific dataset. Inevitably some data will be lost and that is where the anonymization happens. However, what data is lost is now up to an LSTM model which has been heavily used in linguistic analysis. TCP packets

have their own structure, grammar, and alphabet just like a regular language except it's much more simple. In a regular language like English, there are 26 characters, 10 numbers, and a large number of special characters which are combined to make up words and sentences. Due to this complexity, linguistic analysis has moved on from LSTM models to transformer networks for their speed, how they handle word embedding, and their ability to handle longer and more complex sequences. However, this is not a problem for packet data as we only have to deal with the hexadecimal alphabet and packets have a maximum length that is well within the capabilities of an LSTM.

6.2 Dataset

The only dataset I was able to find that could fit this application was CIC-IDS2017 [24] from the University of New Brunswick. The dataset is unfortunately not from a real large corporate network but rather a small network created specifically for the purpose of synthesizing this dataset. The network architecture can be seen in Fig. 6.1. The way this type of dataset is created is by setting up two networks, one comprised of a variety of operating systems on several victim machines and one for the attacking machines. These two networks are connected through some networking architecture that is used to mimic a real networking environment. Then, benign traffic must be collected from the victim machines, which is done through their B-profile system [23]. The B-profile system extracts typical human network behavior and then uses it to generate benign network traffic from the victim machines. Next, the attacking side attacks the victim machines with various types of attacks and records all the network traffic. The problem with this method is again low diversity in the dataset due to the limited size and isolation of the network.

As shown in Fig. 6.1 the capturing server, where the network activity is monitored, is on the victim side. So the attack traffic will end up mainly seeing one IP address, 172.16.0.1, which is the IP where almost all of the attack traffic is coming from. Fortunately botnet and infiltration

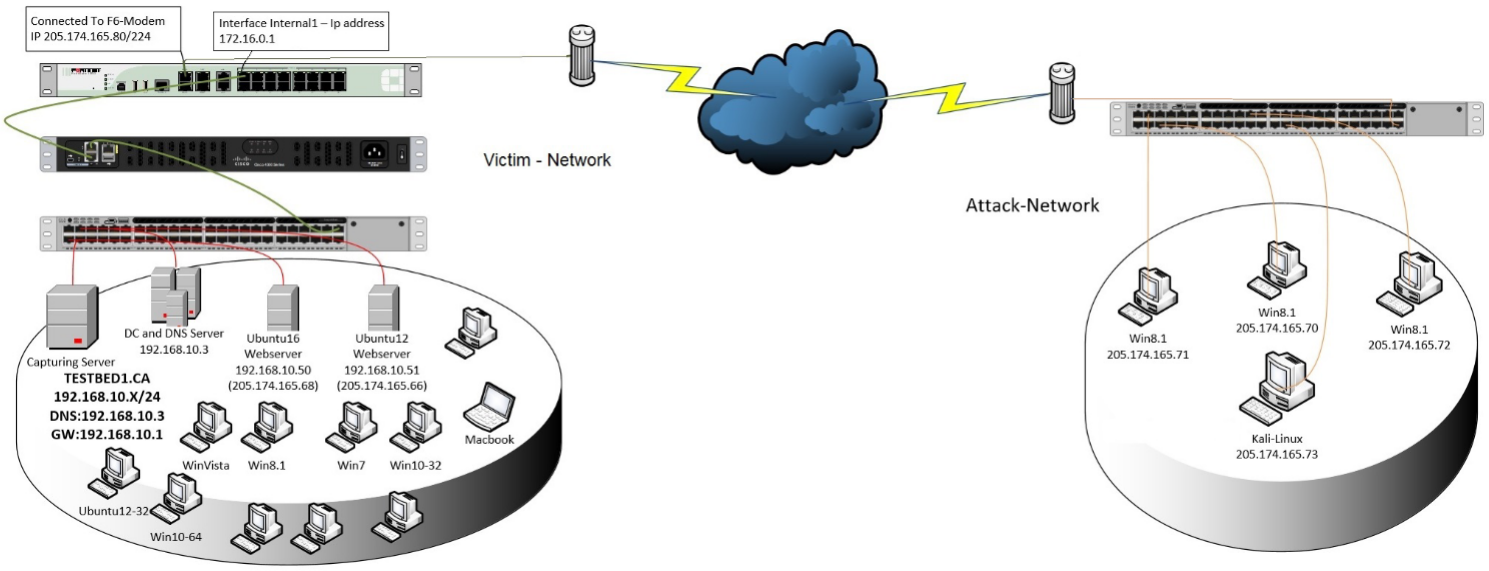

Figure 6.1: CIC-IDS2017 dataset testbed architecture. [23]

attacks include other addresses. This is because for botnets and infiltration attacks the victims themselves are creating the traffic, creating some variability in attacking IPs. However, it is not the same for benign traffic which will see a larger variety of IP addresses. In a real-world scenario, there would be a large number of different attacking IP addresses as well. The limitation of low variability in the attacking IP addresses is one of the reasons that a method like the one described in this section is needed so higher quality, real datasets may be released.

6.2.1 Exploratory Analysis

Like many of the other TCP cybersecurity datasets, the CIC-IDS2017 dataset focused on traffic flows. Fortunately, unlike most other datasets, the original PCAP data was also included in the release. Unfortunately, only the traffic flows were labeled and the packets contained in the PCAP files were not labeled, meaning, a way to map the labeled traffic flows to each packet has to be created. The labeled flows were in 8 different comma-separated value (CSV) files. One CSV file was for Monday but this was not needed since it contained benign traffic only. One CSV file is for Tuesday's traffic, one is for Wednesday's traffic, two are for Thursday's traffic and 3 were for Friday's traffic. Thursday and Friday had multiple CSV files because they

were split up by attack type. The CSV files contained over 80 extracted features for each of the traffic flows but the only features needed were flow ID, source IP, destination IP, source port, destination port, protocol, and label. This is because these are the only ones needed to map the labeled flows in the CSV file to their respective packets in the PCAP file. The flow ID was a unique identifier (UID) that was the tuple source IP, destination IP, source port, destination port, and protocol. However, when duplicate checking was performed on the flow ID column there were many instances of duplicates in this column so extra care will need to be taken to ensure these duplicates are labeled correctly.

The PCAP part of the dataset was created by capturing network traffic over 5 days, being Monday to Friday during working hours. This data was saved to 5 PCAP files that ranged in size from 8 to 13 gigabytes each. On Monday only benign traffic was captured so this will be our benign dataset which totals 11,609,136 packets. During the working hours of Tuesday to Friday, there was a mix of benign and attack traffic. On Tuesday two types of brute force attacks were performed namely, FTP-Patator and SSH-Patator. This resulted in a total of 11,463,288 packets with 43,635 being FTP-Patator and 65,604 being SSH-Patator. On Wednesday Heartbleed and four types of denial of service (DoS) attacks were performed namely, Slowloris, Slowhttptest, Hulk, and GoldenEye. This resulted in a total of 13,694,974 packets with 28,421 being Heartbleed, 36,730 being Slowloris, 31,574 being Slowhttptest, 1,219,986 being Hulk, and 60,774 being GoldenEye. On Thursday five different attacks were performed with three being web attacks and two being infiltration attacks. The web attacks included brute force, cross-site scripting (XSS), and SQL Injection. The infiltration attacks included Dropbox downloads and Cool disk. In total there were 9,240,587 packets with 18,644 being brute force, 5,153 being cross-site scripting, 64 being SQL Injection, and 29,888 being infiltration. On Friday there were three attacks performed namely Botnet ARES, port scan, and distributed denial of service (DDoS) LOIT. In total there were 9,913,145 packets with 6,279 being Botnet ARES, 161,724 being port scan, and 572,593 being DDoS LOIT.

When inspecting Tuesday and Wednesday's CSV files, if all benign flows are filtered

out it can be seen that all attack traffic originates from the IP 172.16.0.1. If the same filtering is applied to Thursday there are two IPs from which the attacks originate, namely, 172.16.0.1 and 192.168.10.8. Upon further investigation, it appears that the web attacks originate from 172.16.0.1 and the Infiltration attacks originate from 192.168.10.8. The IP address 192.168.10.8 is the victim of the Dropbox download infiltration attack. Since this IP is different from 172.16.0.1 it should help with come of the overfitting during traffic generation. It is also included in Monday's benign traffic so it should also help with overfitting on 172.16.0.1 in the classifier. When filtering out all benign traffic on Friday it can be seen that there is much more diversity in its attacking IPs. Having 10 unique IPs, 9 of which are also included in Monday's benign dataset should help with overfitting in the same way as the unique IP in Thursday's dataset.

6.2.2 Prepossessing

The first general prepossessing step is the same for each day. However, how this step is achieved is different for each day. The first step is to first filter the PCAP files so only TCP traffic is left and export the remaining traffic in byte stream format. An example of a TCP packet in byte stream format is shown in Fig. 6.2. Notice there are three sections separated by red slashes. The first section is the Ethernet header which is 14 bytes. The second section is the IP header which is 20 bytes and the third section is the TCP header which is also 20 bytes.

001e4fd4ca2800c1b114eb310800/45000b904465400039060c25170f0418c0a80a0f
/0050c3b2fbf2f0b5fc97fde950100438f1600000

Figure 6.2: Exported TCP packet byte string.

In Fig. 6.3 you can see the example byte string from Fig. 6.2 put into a human-readable format with the various fields and their values that make up each of the three sections of the packet.

```
————————ETH Header————————
Destination Address: 0x1e4fd4ca28
Source Address: 0xc1b114eb31
Ethernet Type: 0x800
————————-IP Header————————-
IP Version: 4
Header Length: 5
Differentiated Services Field: 0x0
Total Length: 2960
Identification: 0x4465
Flags: 0x40
Fragment Offset: 0x0
Time to live: 57
Protocol: 6
Header checksum: 0xc25
IP Source Address: 23.15.4.24
IP Destination Address: 192.168.10.15
————————TCP Header————————
Source Port: 80
Destination Port: 50098
Sequence Number: 4227002549
Acknowledgment Number: 4237819369
TCP Header Length: 80
Flags: 0x10
Window: 0x438
Checksum: 0xf160
Urgent Pointer: 0x0
```

Figure 6.3: TCP packet example in human readable format.

In Fig. 6.4 you can see the size of the different fields that make up the entire TCP packet as well as the three different header segments namely the Ethernet, IP, and TCP headers. It was decided that the recreation of the TCP payload would be too much for the algorithm to handle due to the fact that it is encrypted, making it seem random. Due to this, during prepossessing, the TCP payload and the options field were dropped. The options field is mostly used for setting up the connection and I did not find any popularized attacks that relied on its use. So for the sake of simplicity, it was dropped. Having a fixed-length also increased the speed of the algorithm considerably, which will be talked about more in Section 6.6.

To filter out all protocols except TCP and export in byte stream format, Wireshark [7]

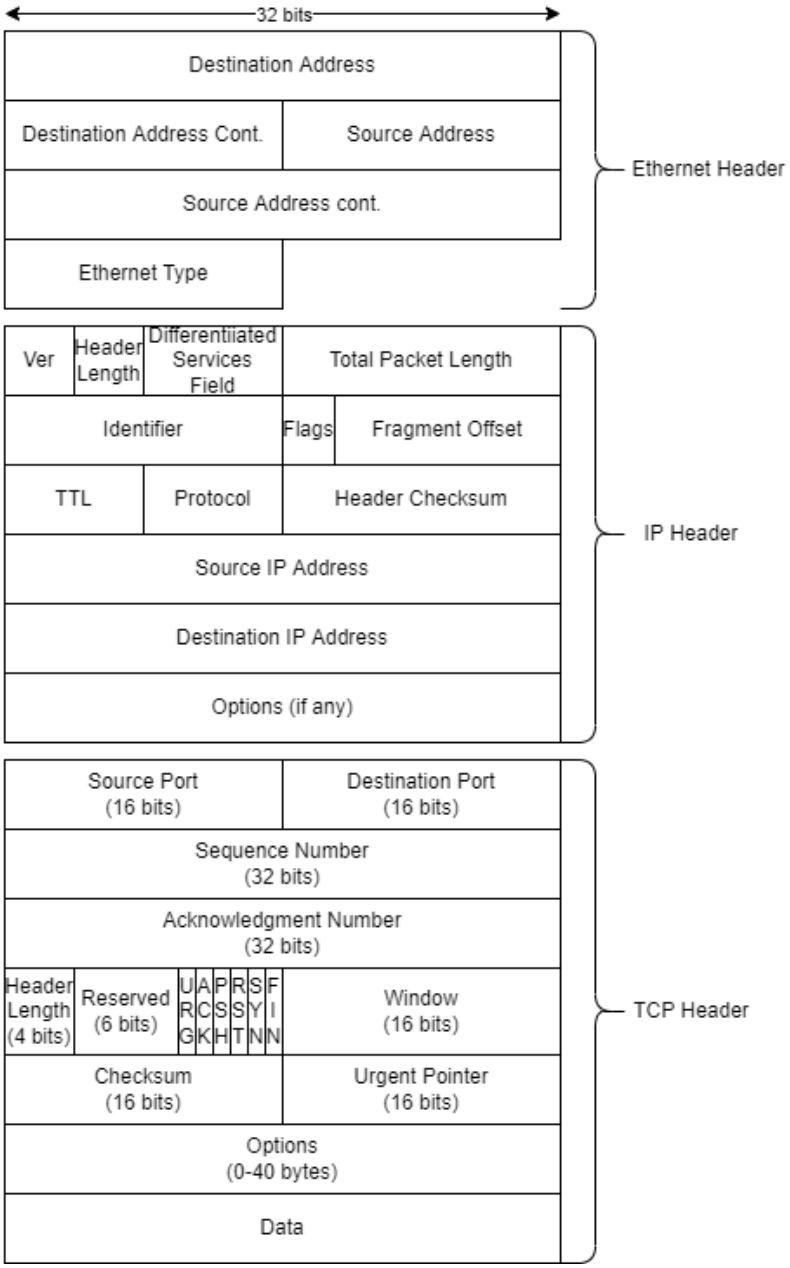

Figure 6.4: TCP Packet Structure.

needed to be used. However, there were immediate problems with this. The first problem was for Wireshark to load a 10+ gigabyte, file it took several hours. The second problem was once the filter for TCP was applied and exporting the packet dissections was attempted, it became too much for Wireshark and it would crash. So the only solution was to first make the PCAP files smaller using Splitcap [18]. When using the PCAP files with Splitcap it kept

giving a file type error and the solution ended up being to load the PCAP files into Wireshark and then just save them again as a PCAP file which fixed the Splitcap loading problem. Once splitting and filtering were done, the packet dissections were exported in byte stream format using Wireshark. This resulted in large JSON files containing more information than was needed.

The next part of the prepossessing was extracting and mapping the packet byte streams from the JSON file to the TCP flow labels provided with the dataset in CSV format. The problem is that each TCP flow contains multiple packets, and the UID tuple (source IP, destination IP, source port, destination port, and protocol) is not unique. Meaning, that some packets have multiple labels and could be labeled as both benign and malicious. In the case where there are multiple labels but one has a majority, the majority label will be kept. In the case where there is no majority label or a tie, the packet will be dropped since it can not be determined if it is benign or malicious. This is, in general, how prepossessing was done; however, each day had to be done slightly differently.

First, Monday was the only day with exclusively benign traffic, so no splitting was performed. It also had more than enough benign traffic samples, so only malicious traffic was extracted from Tuesday-Friday. Second, some days were split using Splitcap, and some were not. It depended on where the malicious traffic was originating from. Some days malicious traffic only originated from 172.16.0.1, and some days it originated from multiple IP addresses, so splitting could not be performed. Third, each day had different Wireshark filters applied depending on what traffic types were included in the PCAP file. Nevertheless, the end goal was the same, to filter out all traffic types except TCP. Finally, the biggest differences came in the way the labeling was done. Different algorithms were used depending on the number of labeled flows contained in the CSV file and the number of packets that had to be labeled. These algorithms all aim to transfer the labels from the flows to the packets but are designed differently to be more efficient under different circumstances. More detailed information on preprocessing for each day can be found in Appendix A

6.3 Character-Level Long Short-Term Memory Models

The two LSTM models in this section are similar to the ones in Section 4.2 except they generate TCP packets instead of URLs. Similarly, they each have an embedding layer, a two-layer LSTM, a dropout layer, and a fully connected layer. The only structural difference is that another dropout layer has been added between the two LSTM layers with a dropout probability of 0.2.

6.4 Training

In this section, the training process is described, which is almost the same as in Section 4.4 except the samples all have a constant length, so no padding token is needed for batching. The dataset for the benign data was also much larger than in Chapter 4. This led to a different validation process being needed during the benign generators training. Instead of validating at the end of each epoch, validation was performed after every 8000 batches. This led to a lower final validation loss and higher quality synthetic samples. A detailed description of the training process follows.

The goal of training is to train the character-level LSTM models to generate data with the same characteristics as the original TCP packet data but not generate the same data. The first step is splitting the data into train and validation sets; 90%, and 10%, respectively. There is no test set because character-level LSTM models cannot be tested by comparing inputs to generated sequences. It will be shown later in Section 6.7 how the models are tested after training.

The only input to the models is the start token, and then they generate packets one character at a time based on the probability of what the model thinks the next character should be. This process will be more clear when explained in the context of training batches. In a batch of size one, take a sample and make two arrays, one for input and one for target, each being of size one longer than the length of the sample (length of n+1). The input array has the start token

placed in the first position of the array and then each character of the sample in the rest of the cells. The target array has the characters of the sample placed in the first n cells and the end token placed in the last cell (n+1). This creates an offset between the input and target arrays. This is because the LSTM is trying to predict the next character; therefore, if the input is the start token, then the next character (target) is the first character in the TCP packet and onward. The batching process here is basically the same as in Section 4.4 except the samples all have the same length since the payload was not used. Due to this, no padding token is needed, which also makes the batching process much faster. Batch shuffling was also done at the beginning of each epoch. This means that the order of the training samples was shuffled after each epoch and the batches recreated. More information about the batching process is given in Section 6.6.

The training function uses cross-entropy loss and Adams optimization with a learning rate of 0.001, β_1 of 0.9, and β_2 of 0.999. It has a batch size of 128, and to ensure that we do not run into the exploding gradient problem with really long URLs, the gradient norm is clipped with a max norm of 5. Early stopping using the validation loss was the same for the malicious LSTM model but had to be implemented differently for the benign LSTM due to the size of the dataset.

The benign dataset had almost 9.2 million packets which is about eight times larger than the malicious dataset. This meant that, on average, the benign LSTM generator model would converge after about 3 epochs. However, in experiments, it was noticed that many of the samples the benign LSTM model generated were not very good. They had incorrect lengths and other problems that the samples generated by the malicious LSTM model did not have.

This led to the execution of an experiment where the benign LSTM model was trained with different size samples of the benign dataset. It was trained with 250,000, 500,000, 1,000,000, 3,000,000, 6,000,000, and 9,156,151 benign samples selected randomly from the original benign dataset. What was found was that the number of epochs and lowest validation loss would decrease from 250,000 samples to 3,000,000 samples but then would start to increase when 6,000,000 and 9,156,151 samples were used. This led to the conclusion that having such a

large number of samples per epoch and only calculating the validation loss once per epoch was most likely causing the model to miss the lowest loss value and start overfitting before early stopping could stop training.

Due to this, it was decided that a different validation procedure may lead to better results. This validation procedure calculated the validation loss after a certain number of batches were processed rather than only after each epoch. However, this meant that the number of batches between validation steps needed to be determined. There was a total of about 64,000 training batches and 8,000 validation batches, so it was decided the minimum would be to validate every 8,000 training batches since any lower and the validation step would take longer than a training step. With this decided, an experiment was set up that tested validation after 32,000, 24,000, 16,000, and 8,000 batches. The results showed that validating every 8,000 training batches achieved the lowest validation loss. This showed that the assumption was correct about the LSTM model overshooting the lowest validation loss when it was only calculated after every epoch. So, the model was set up to validate after every 8,000 batches, solving the problem of poor quality synthetic samples. This also makes sense since the malicious LSTM model only had about 8,000 training batches, so when it validated every epoch, it would not have run into the same issues. For information about the hardware environment see Appendix B.

6.5 Generating TCP Packets

The steps to generate the TCP packets were kept the same as in Section 4.5. Now a natural question would be why keep the start and end tokens if the packets are fixed length. The start and end tokens were kept as a way to detect malformed TCP packets. Keeping the tokens allows the model to continue predicting past the 108 character length of the TCP packet. If a packet is longer than 108 characters, it is discarded from the synthetic dataset. If the start and end tokens were not kept, and the model could only generate 108 character length strings, then the model may be stopped in the middle of creating a malformed packet. These malformed

packets would then be very hard to find and remove from the synthetic dataset. However, since the model can generate any length it wants, it is easy to find and remove the malformed packets from the synthetic dataset if the model generates a packet that is too long.

6.6 Batching

In this section, both the batching and batch shuffling processes are described. The original batching process is optimized for this chapter by caching steps that are the same each time a batch is created. The padding of sequences in a batch is also removed since the packets have a constant length in this chapter. This significantly speeds up the batching process from about 28 minutes to 16 seconds, allowing batch shuffling to be implemented without adding too much time to training. Batch shuffling happens at the beginning of each epoch and changes the order of all the training samples, then remakes the batches. This improves the overall performance of the models. A detailed description of what was just described follows.

To reduce overfitting and improve the generalization performance of the models, it was decided it would be a good idea to implement batch shuffling at each epoch rather than only creating the batches once at the beginning of the program. However, in Section 6.2.1, we saw that there are about 1.1 million malicious packets, which makes shuffling all training samples and recreating the batches at each epoch very time-consuming.

The original batching function I was using took 28 minutes to complete, which takes way too much time if it is going to be done at the beginning of each epoch. This meant the batching function had to be optimized. The first thing I did was create a separate python file that would offload some of the steps that had to be repeated each time batches were created by doing those steps and saving the resulting arrays to the SSD so they could be loaded quickly. Two steps that were not a part of the batching function but were added to this were loading the hex streams from the JSON data from the PCAP files and encoding them into integer representations. Originally these steps were done at the beginning of the python notebook but were still cumbersome to do each time the IDE was reset and took about a minute. So the first part of the batching

function was to take the list of packets and make the input and target arrays by adding the start character to the beginning of the input array and the end token to the target array. This was removed and added to the new data loading program, which saved a significant amount of time by not having to be repeated each time new batches had to be made. The input and target arrays were then saved to the disk. This is so they could be quickly loaded into RAM at the beginning of the training program and accessed quickly every time the batches had to be shuffled.

Next was optimizing the batching function itself. I realized that, unlike the DNS and URL data, since we got rid of the TCP payload, all the strings would be constant length. This meant that they did not have to be padded. So the padding token was removed from the character dictionary, and the padding steps were removed from the function. This also meant that in the batching function, they did not have to be sorted by sequence length, and the sequence lengths did not have to be recorded at all, which removes several steps. The next part to optimize was how the batches were turned into tensors. In the original batch function, two tensors were created, one for the input batch and one for the target batch. These two tensors were of size batch size by the length of the longest sample. Originally, the start token was added to the first spot in the input array, and the end token would be added to the spot after the length of each string in the target array. This was no longer needed since the start and end tokens were already added to the TCP samples before being saved to the SSD, which was discussed in the last paragraph.

The way the data was copied to the tensor used a nested for-loop which was extremely costly. This was no longer needed as rows (containing one TCP packet each) could just be selected from the arrays that were saved to the SSD, and then a tensor could be made directly from them. Overall, this improved the batching process from about 28 minutes to 16 seconds, allowing for batches to be reshuffled at each epoch without adding too much overhead. This also allowed the removal of the packing and padding parts from the LSTM's forward function, which allowed it to process about 47 training batches per second, up from 44 batches per second.

6.7 Evaluation

This research had two goals: to create data that was characteristically similar to the original in terms of classification performance and for that data to be different enough (anonymized) from the original data so that it could be made publicly available. To evaluate these requirements, three tests were performed. The first test attempted to classify the TCP packets into "malicious" or "benign" to satisfy the first requirement. The second test checked the similarity between the real and synthetic TCP packets to satisfy the anonymization constraint. Finally, the third test compared the high-level structure of the real and synthetic datasets to reinforce the findings of the first two tests in terms of characteristic preservation and anonymization.

For these tests to be performed, 50,000 malicious and 50,000 benign TCP packet samples were generated. Then any TCP packets with a length longer or shorter than 108 characters were removed, and any duplicates within each set were removed. This resulted in 49,727 synthetic benign samples and 49,924 synthetic malicious samples.

6.7.1 Classification Test

The classifier is another LSTM model but modified to do binary classification. This test for characteristic preservation will involve giving the network a list of malicious TCP packets (class 1) and a list of benign TCP packets (class 0) and asking it to classify them into one of the two classes. For this test, two classifiers are trained, one on the real benign and malicious datasets and one on the synthetic benign and malicious datasets. These two classifiers are then tested on both the real and the synthetic data. So overall, there will be four parts to this test: train on real, test on real (TRTR), train on synthetic, test on synthetic (TSTS), train on real, test on synthetic (TRTS), and train on synthetic, test on real (TSTR). The performance of these classifiers is then compared, and if the classifiers have similar performance, we can say the real data's characteristics have been preserved in the synthetic dataset. Several metrics were used to compare the performance, including accuracy, recall, precision, specificity, and confusion

matrices. The classifier was trained with a learning rate of 0.001, binary cross-entropy with logits loss, a batch size of 128, and a two-layer LSTM with a hidden size of 512. It was allowed to go for a maximum of 50 epochs, but early stopping was set to 3; therefore, training will stop if the validation loss does not improve after 3 epochs.

The data split used was 65% train, 10% validation, and 25% test when doing TRTR and TSTS. When TRTS is performed, the entire synthetic dataset is used to test it, and since the classifier has not seen the synthetic data during training, there is no leakage. The same is true for TSTR, where the entire real dataset will be used for testing since it was not used during training.

The real malicious dataset has 1,181,103 TCP samples, and the real benign dataset has 9,156,151 samples. This created a large imbalance in the dataset, which would end up causing a skew in the classification results for the classifier trained on only real data. This would make a comparison to the classifier trained on synthetic data difficult since the synthetic datasets are balanced. Foreseeing the problems that an imbalanced dataset could cause, it was decided to randomly select 1,181,103 TCP packets from the real benign dataset, giving it the same size as the real malicious dataset. Removing part of the real benign dataset should not cause any problems with the Levenshtein or UMAP tests due to the repetitive nature of TCP header data. Making this part of the dataset smaller also benefits the Levenshtein similarity test, as discussed below.

6.7.2 Levenshtein Similarity Test

The Levenshtein ratio was used to evaluate how similar a synthetic TCP packet is to a TCP packet in the real dataset. Using the Levenshtein distance [17], one can measure the difference between two sequences. The Levenshtein distance between strings represents the number of single-character edits (insertions, deletions, and substitutions) necessary to convert one string into another. The Levenshtein ratio uses the Levenshtein distance to calculate the percent similarity between two strings or, in this case, two TCP packets. For example, two strings are

the same if they have a Levenshtein ratio of 100%. Using the Levenshtein ratio, every synthetic malicious TCP packet is compared to every real malicious TCP packet. Every synthetic benign TCP packet is compared to every real benign TCP packet, and the highest similarity ratio is recorded. These are then plotted on two histograms to visually represent the two similarity distributions. Then based on a user's similarity tolerance, packets above a certain similarity ratio can be removed.

The large size of the benign dataset posed a problem when performing this test. Since every synthetic benign sample has to be compared to every real benign sample, a nested loop comparison structure has to be performed. This was extremely slow and was estimated to take 58 days to complete, which is way too long. To overcome this, it was decided to multi-thread the process taking advantage of 20 threads which left 4 threads, so the computer was still functional while the program was running.

To take advantage of multi-threading, the synthetic benign sample list of 49,727 was divided into 20 smaller lists (one for each thread). Then each of the samples in those smaller lists would be compared to all the samples in the real dataset, and the highest similarity would be recorded. When the program was completed, the results from each of the 20 threads would then be combined again, giving the highest similarity for each of the synthetic packets. Once the modifications were complete, the program took about 4 days to complete. This is a massive improvement but still too long considering how many times it needs to be run for experimental purposes. The final improvement was to cut down the real benign dataset to 1,500,000 samples from 9,156,151, much like was done in Section 6.7.1. This seemed like a reasonable number because of TCP data's repetitive nature when the payload is removed. This reduced the benign dataset's processing time from 4 days to about 15 hours and 30 minutes. The same multi-threaded process was used, without downsizing the real datasets, when doing the Levenshtein ratio tests on the synthetic malicious dataset in this chapter and both synthetic datasets in Chapters 4 and 5.

6.7.3 Uniform Manifold Approximation and Projection

The Last test used uniform manifold approximation and projection (UMAP) to dimensionally reduce the TCP packet data from its 24 components which can be seen in Fig. 6.3 to 2 components so it can be plotted on an x and y-axis for visualization. UMAP is similar to t-distributed stochastic neighbor embedding (t-SNE) [26] except that the transform from higher to lower dimensions is learnable. This subtle difference allows us to train it on real data only, then apply the transform to an unseen test set of the real data and the synthetic data. The results of the transformation of the test sets will be plotted in a graph in an overlapping fashion. If the characteristics of the data were preserved in the synthetic dataset, the results should be a bunch of data points clumped together with the synthetic data points overlapping and having roughly the same grouping as the real data points. In this, the clumps will also represent the modes of the data, so if the synthetic data has a similar spread to the real data, then this would show there was no mode collapse in the model. On the other hand, if the data points are really tightly grouped and not spread out or centered in only one part of the graph, this would show there was mode collapse in the model. In another case, the synthetic data points could not follow the pattern of the real data at all, and this would show that the characteristics of the data were not captured effectively.

This test is more for characteristic preservation, but seeing the high-level structure of the data can reinforce findings from the Levenshtein test. When looking at the graph in terms of anonymization, the goal would be to look for a pattern where the major modes are followed, but some separation in the points can also be seen. A good result would be a graph where some but not all points overlap and there is some randomness in the data. A poor result would be where all the synthetic data points completely cover the real data points until no real data points can be seen.

The hyperparameters of the UMAP model were just left as default. This is because UMAP is normally used to try to separate and cluster data from a single dataset, and depending on how the hyperparameters are set, different groupings or modes can be seen. However, in this

case, we are not looking to separate data. We just want to see how the real and synthetic data compare when they are reduced to two dimensions. So, if the hyperparameters are set to default during training, any information lost during the transform will also be lost when the transform is applied to the synthetic data. This results in two sets of data points that can be compared on equal terms.

6.8 Results

6.8.1 Malicious TCP LSTM model

The first model that was assessed was the malicious LSTM that generates the synthetic malicious TCP packets. The training took place over 16 epochs, with the lowest validation loss recorded at epoch 13. The model converged to a training loss of about 0.617 and a validation loss of about 0.613. A visual representation of this can be found in Fig. 6.5.

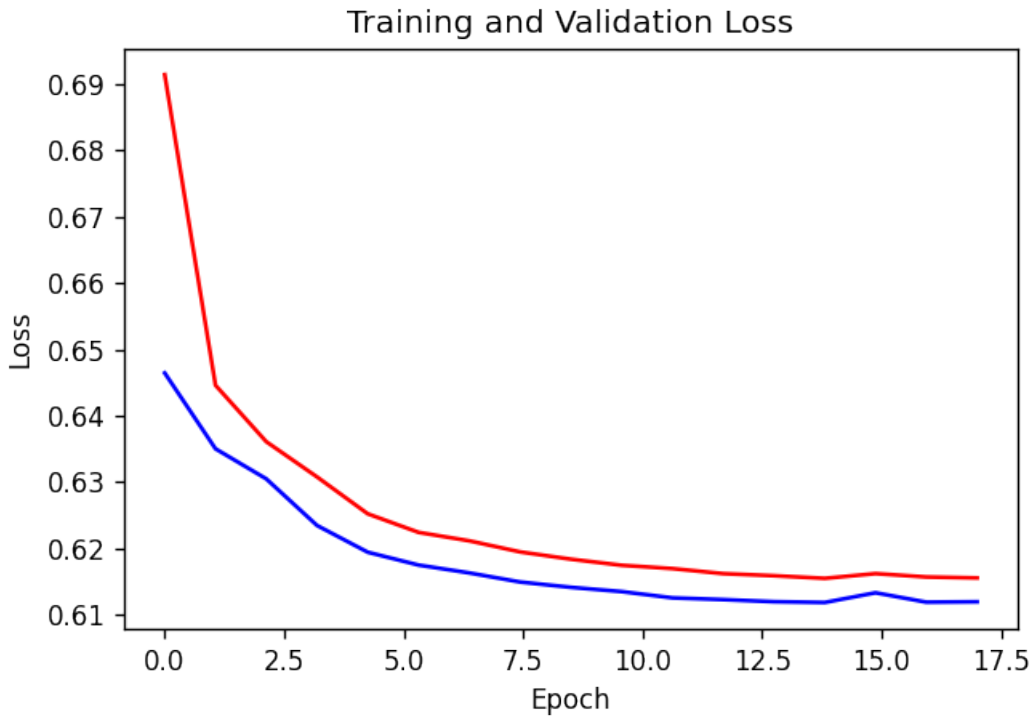

Figure 6.5: Malicious TCP LSTM model training graph.

The Levenshtein ratio histogram shown in Fig. 6.6 shows how similar a synthetic malicious TCP packet is to TCP packets in the real dataset. Looking at the histogram, it can be seen that most of the data points are between 0.82-0.95. The average value for this histogram is 0.875. This indicates that, on average, a synthetic data point is 87.5% similar to a data point in the real dataset. The maximum value that the histogram got was 0.963, and the minimum value the histogram got was 0.787.

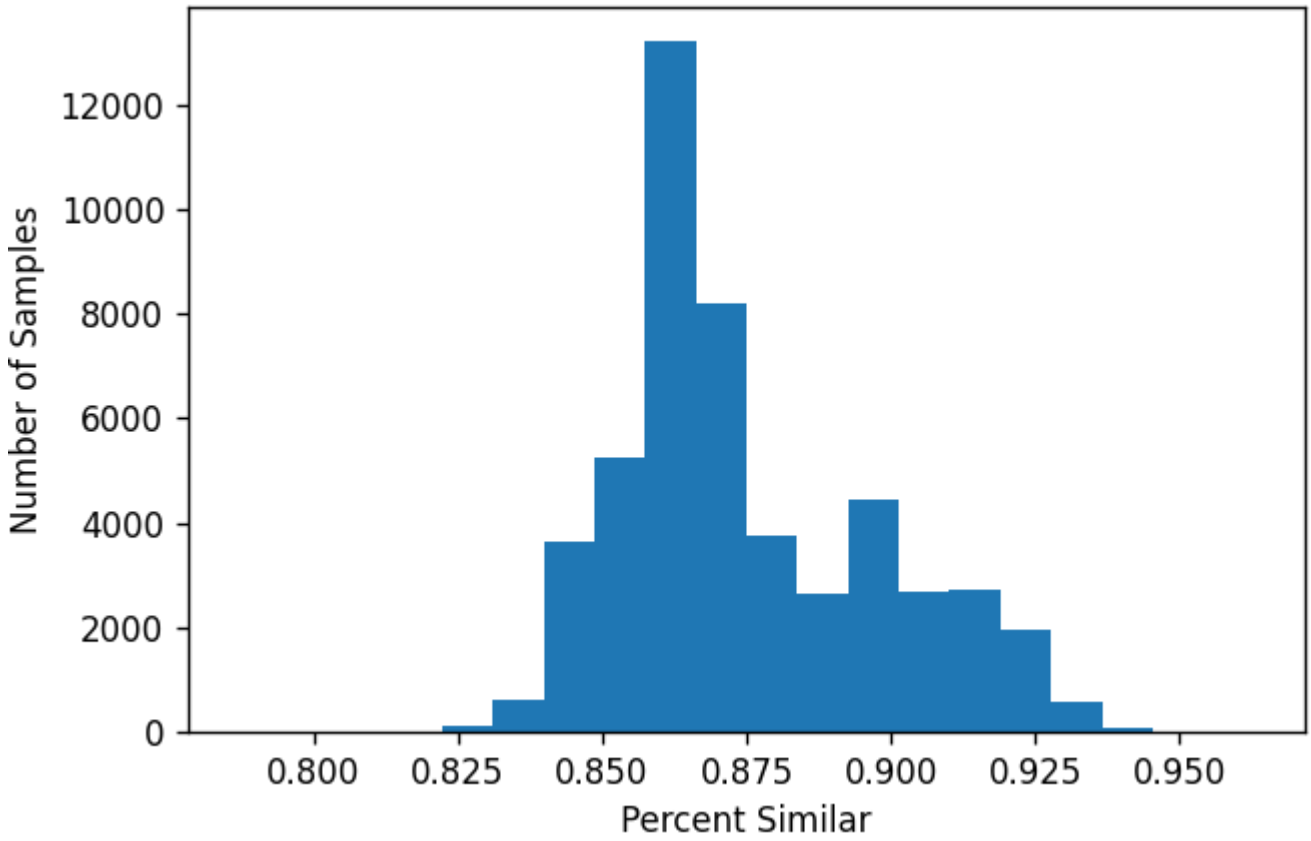

Figure 6.6: Similarity of synthetic malicious TCP packets to real malicious TCP packets.

6.8.2 Benign TCP LSTM model

The next model to be evaluated was the LSTM which generates benign TCP packets. The training period was 7 epochs, with the lowest validation loss recorded at epoch 4. In training, the model's training loss converged to a value near 0.590, and its validation loss converged to a value of about 0.573. This is illustrated in Fig. 6.7.

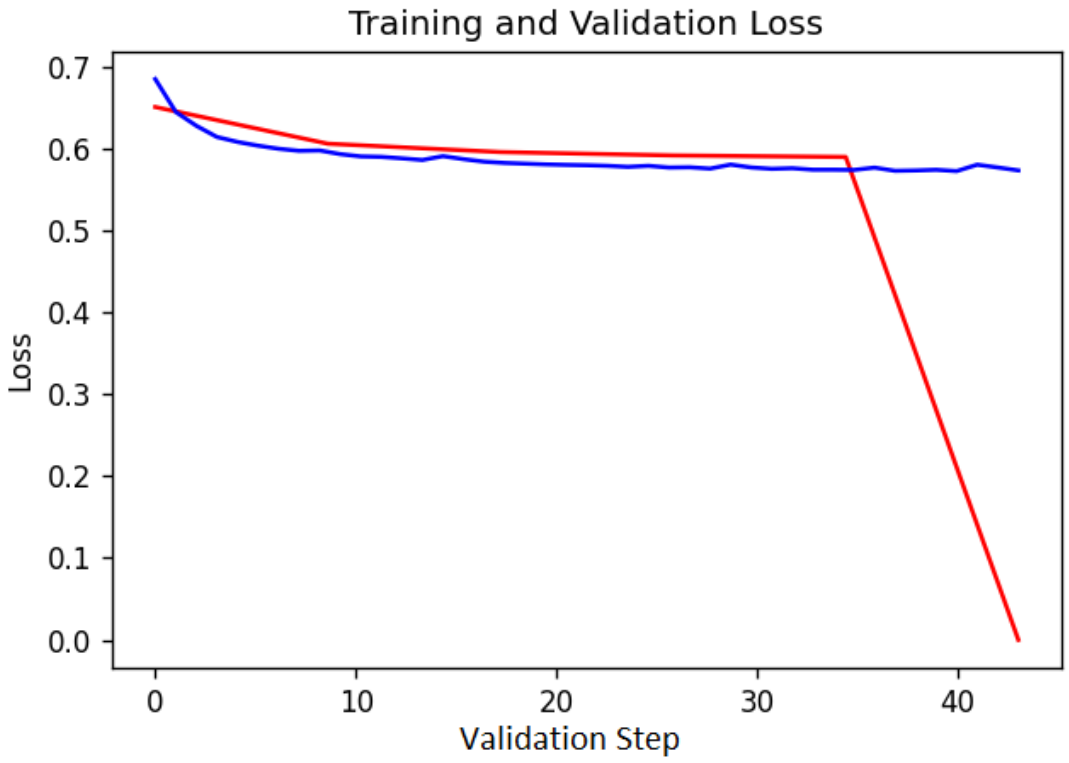

Figure 6.7: Benign TCP LSTM model training graph. Here the x-axis represents the validation step rather than the epoch since validation was performed more than once per epoch.

According to the histogram shown in Fig. 6.8, it can be seen that most of the data points lay between 0.75 and 0.96. This gives the histogram an average value of 0.8695. As a result, on average, a synthetic data point has a 86.95% similarity to a real data point. A maximum value of 0.9722 was obtained in the histogram, and the minimum value was 0.7037.

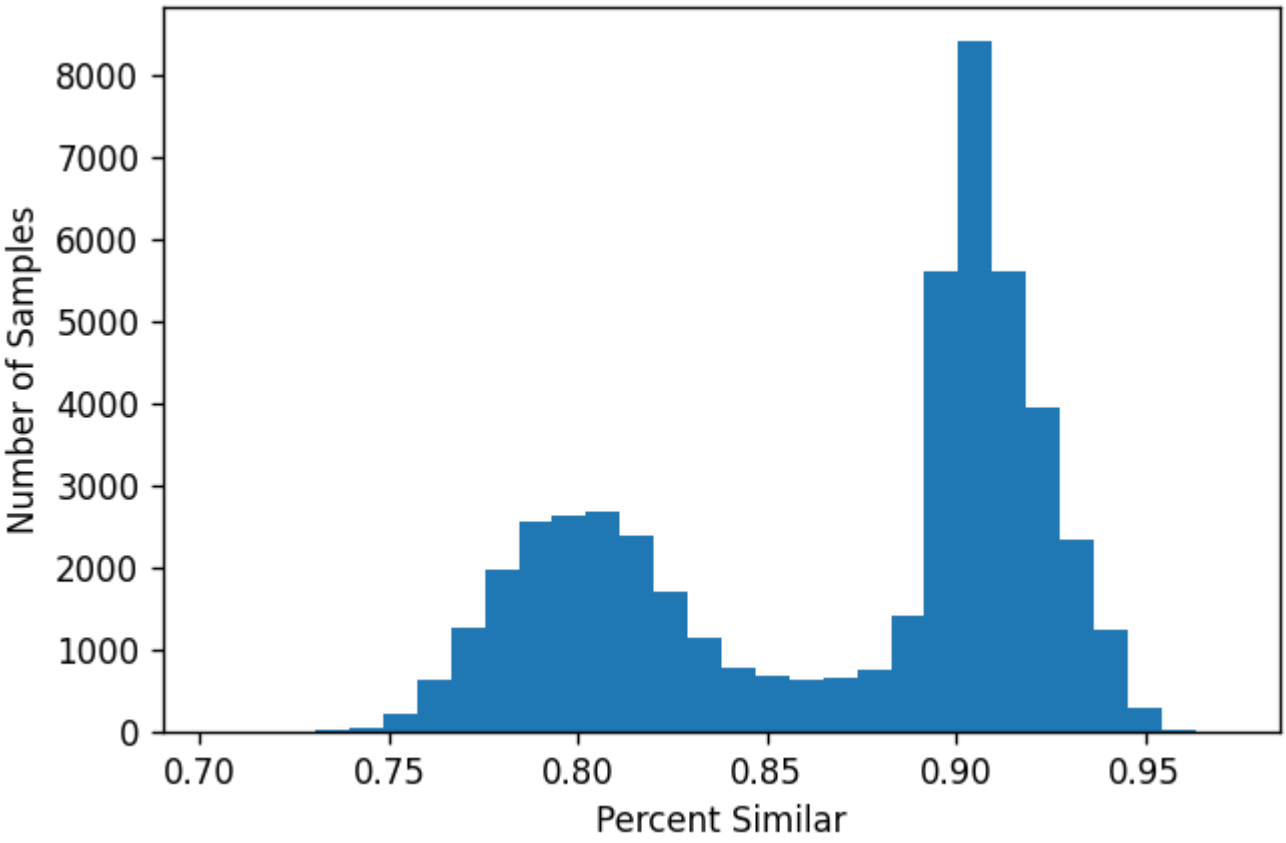

Figure 6.8: Similarity of synthetic benign TCP packets to real benign TCP packets.

6.8.3 Classifier Trained on Real Data

The classifier trained on real data was the next model to be evaluated. There were 10 epochs during the training period for this model, with the lowest validation loss recorded at epoch 7. The model showed a very low training loss, converged to around 0.00005. Additionally, the validation loss was extremely low, converging to about 0.0000000078. This can be seen in Fig. 6.9. After the model was trained, the model was tested twice, once using the real test set and once using the synthetic dataset. The results of the test sets can be seen in Table 6.1.

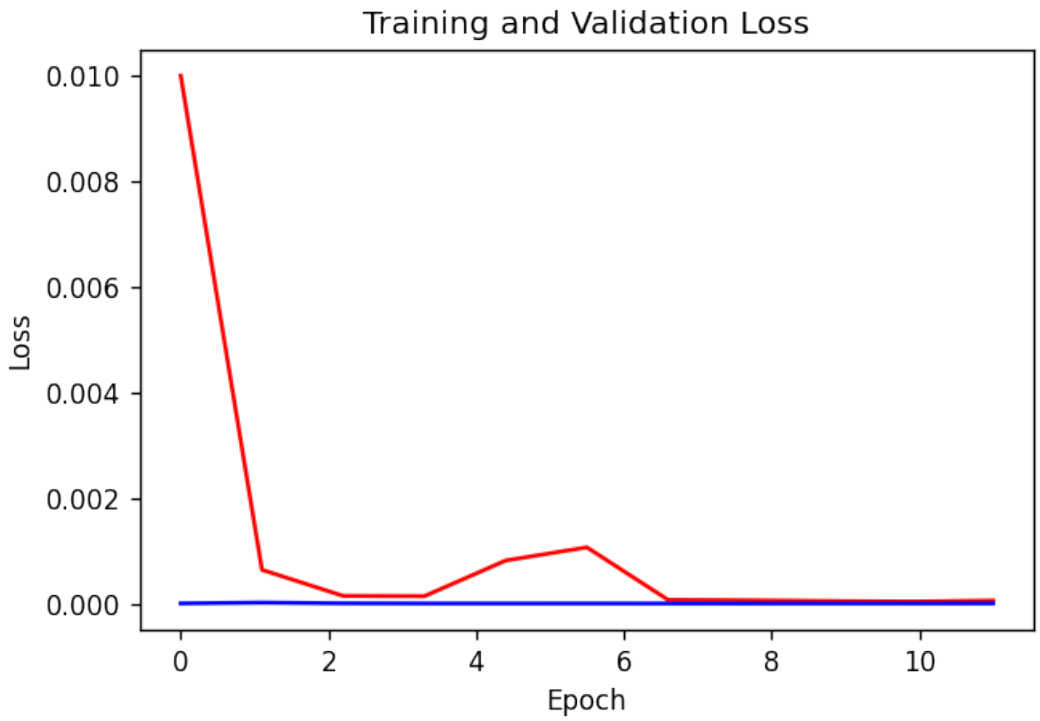

Figure 6.9: Classifier trained on real data training graph.

Metrics	TRTR Values	TRTS Values
Accuracy	0.9999	1.0
Recall	0.9999	1.0
Precision	1.0	1.0
Specificity	1.0	1.0

Table 6.1: Classifier Trained on Real Data Test Results

The classification results of the classifier trained on real data can also be seen in greater detail by examining the confusion matrices. In Fig. 6.10, you can see the confusion matrix for the classifier trained and tested on the real dataset. Here it can be seen why the classification metrics are so high in Table 6.1 as only 3 TCP packets have been misclassified out of 590,464 samples. Next, in Fig. 6.11, you can see the confusion matrix for the classifier trained on real data and tested on the synthetic dataset. Here it can be seen that it misclassified 0 out of 99,584 synthetic data points.

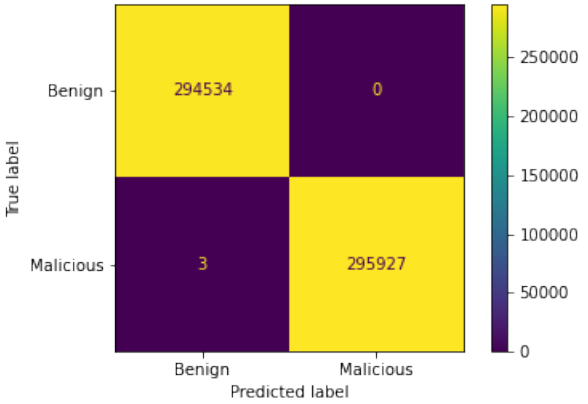

Figure 6.10: TRTR classifier confusion matrix.

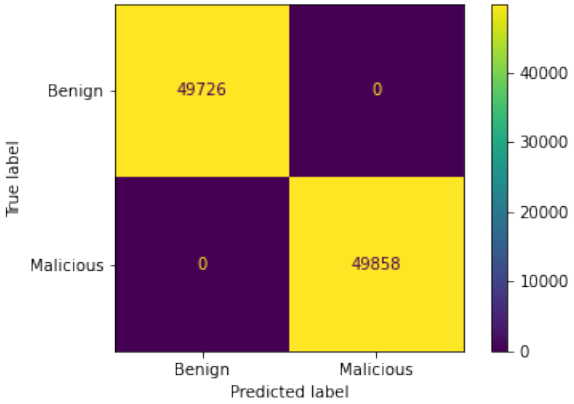

Figure 6.11: TRTS classifier confusion matrix.

6.8.4 Classifier Trained on Synthetic Data

Next, the synthetic data classifier was evaluated. There were 38 epochs during the training period for this model, with the lowest validation loss achieved at epoch 35. During training, the model showed a very low training loss, converging to about 0.00000000034. Furthermore, the validation loss was also very low, converging to 0.0. This can be seen in Fig. 6.12. After the model was trained, the model was tested using the generated test set. The results of the test sets can be seen in Table 6.2.

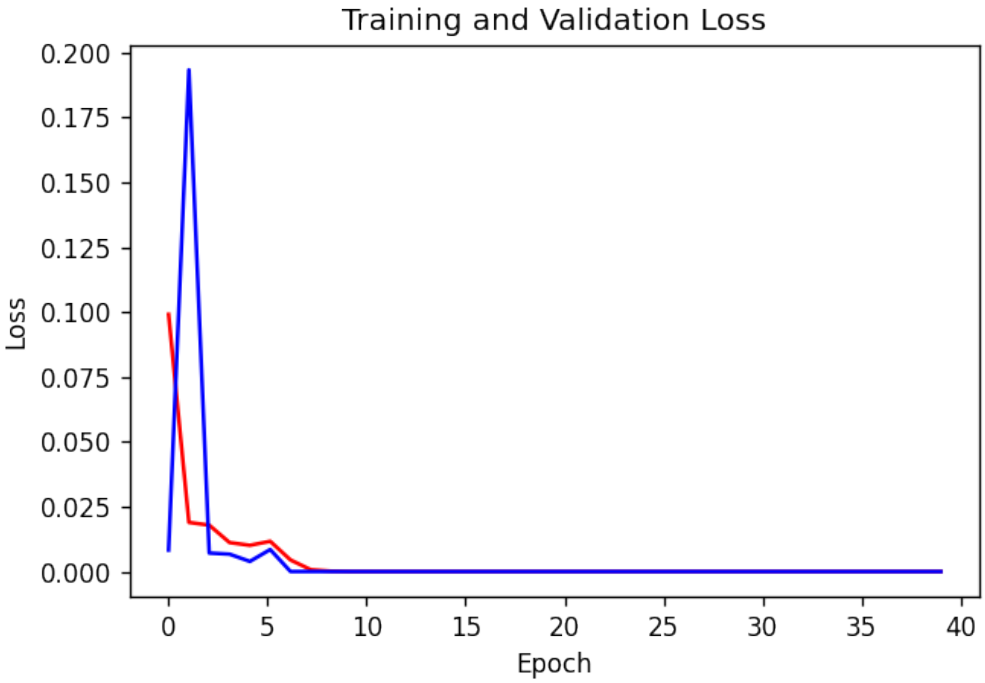

Figure 6.12: Classifier trained on synthetic data training graph.

Metrics	TSTS Values	TSTR Values
Accuracy	0.9999	0.9999
Recall	1.0	0.9999
Precision	0.9999	0.9999
Specificity	0.9999	0.9999

Table 6.2: Classifier Trained on Synthetic Data Test Results

The classification results of the classifier trained on synthetic data can be seen in greater detail by examining the confusion matrices. In Fig. 6.13, you can see the confusion matrix for the classifier trained and tested on the synthetic dataset. Here it can be seen why the classification metrics are so high in Table 6.2 as only one TCP packet has been misclassified out of 24,832 samples. Lastly, in Fig. 6.14, you can see the confusion matrix for the classifier trained on synthetic data and tested on the real dataset. Here it can be seen that it misclassified 94 out of 2,362,112 real data points. This is slightly more than TSTS in terms of the number of misclassifications, but there are a lot more test samples.

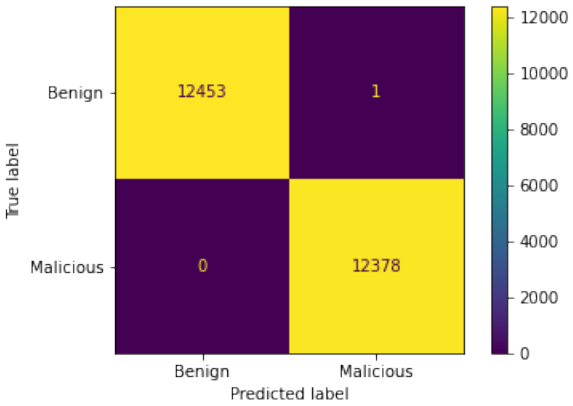

Figure 6.13: TSTS classifier confusion matrix.

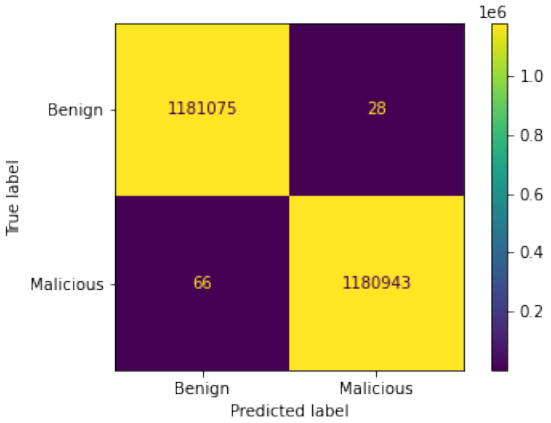

Figure 6.14: TSTR classifier confusion matrix.

6.8.5 UMAP

In this section, two tests were performed, one that compares the synthetic benign samples to the real benign samples and one that compares the synthetic malicious samples to the real malicious samples. For this experiment, only a smaller amount of the real benign data can be used due to computational restrictions. If all 9 million real data points are used, the algorithm uses a huge amount of RAM and crashes. So it was decided to make the amount of real benign data used the same as the amount of real malicious data used in Section 6.7.1. This means 1,131,103 real benign data points were used as a training set for the benign UMAP model with 50,000 set aside as an in-sample test set and 1,131,103 real malicious data points as a training set for the malicious UMAP model with 50,000 set aside as an in-sample test set. 50,000 real samples as a test set were chosen to match the approximately 50,000 synthetic samples that will be used for testing. Due to size restrictions of the graph produced by UMAP, if too many real data points are used for testing, they overlap too much, and it just looks like a blob where it is hard to see any patterns emerge. So 50,000 seemed like a reasonable choice to balance interoperability of the results graph with a big enough sample size to draw conclusions.

To train the UMAP models, first, both sets of real TCP samples were dissected into their packet attributes like in Fig. 6.3. Those attributes were then converted from their hexadecimal

values into integers and then scaled by removing the mean and scaling them to unit variance. This process was also done for the 50,000 synthetic benign TCP packet samples and the 50,000 synthetic malicious TCP packet samples that make up the two out of sample test sets for the UMAP models. In Fig. 6.15 and Fig. 6.16 you can see the results of the UMAP dimensionality reduction for the benign and malicious TCP data, respectively.

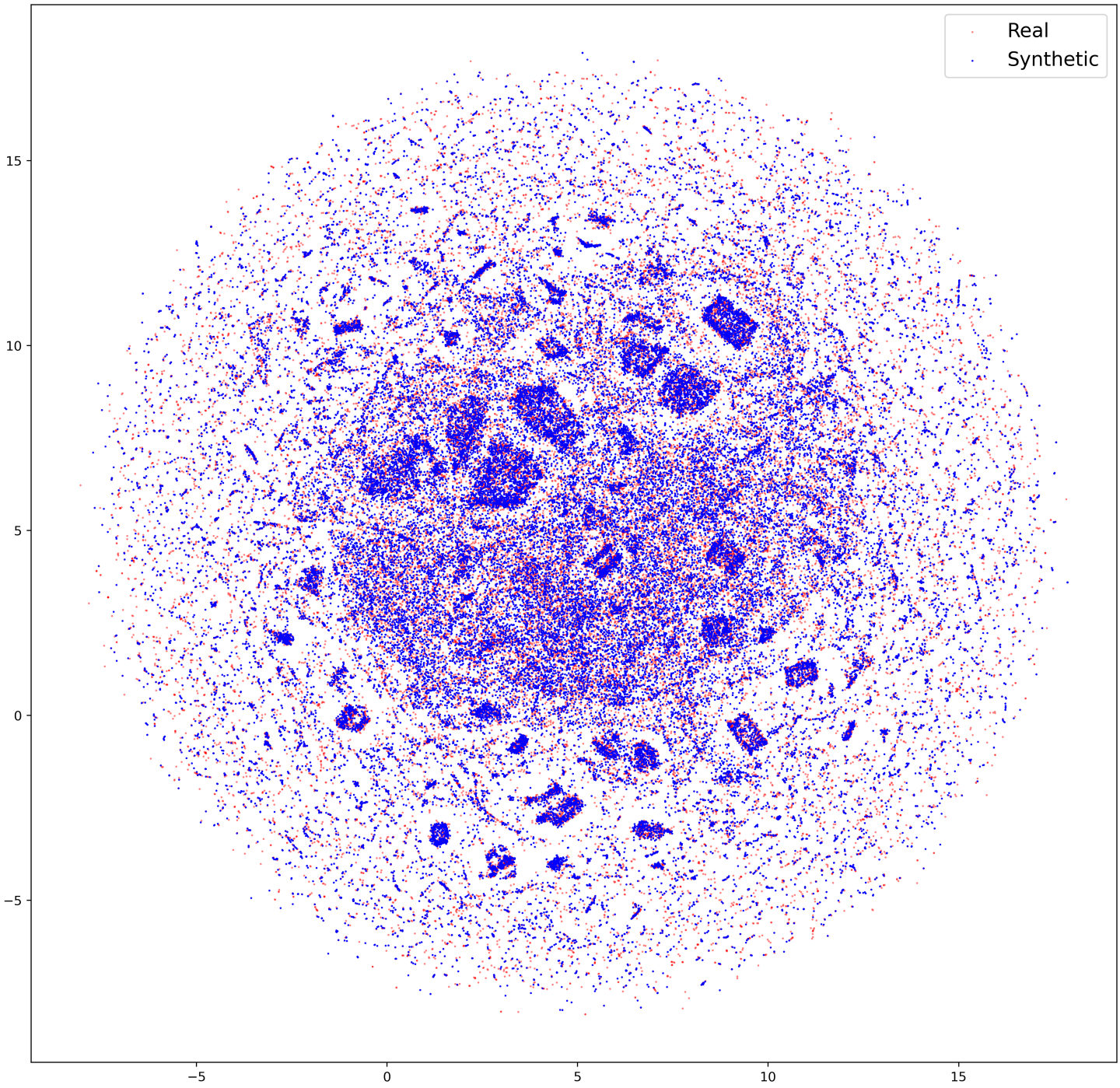

Figure 6.15: UMAP dimensionality reduction of the real and synthetic benign TCP packet data, which maps multi-dimensional datasets to two dimensionless attributes to help visualize.

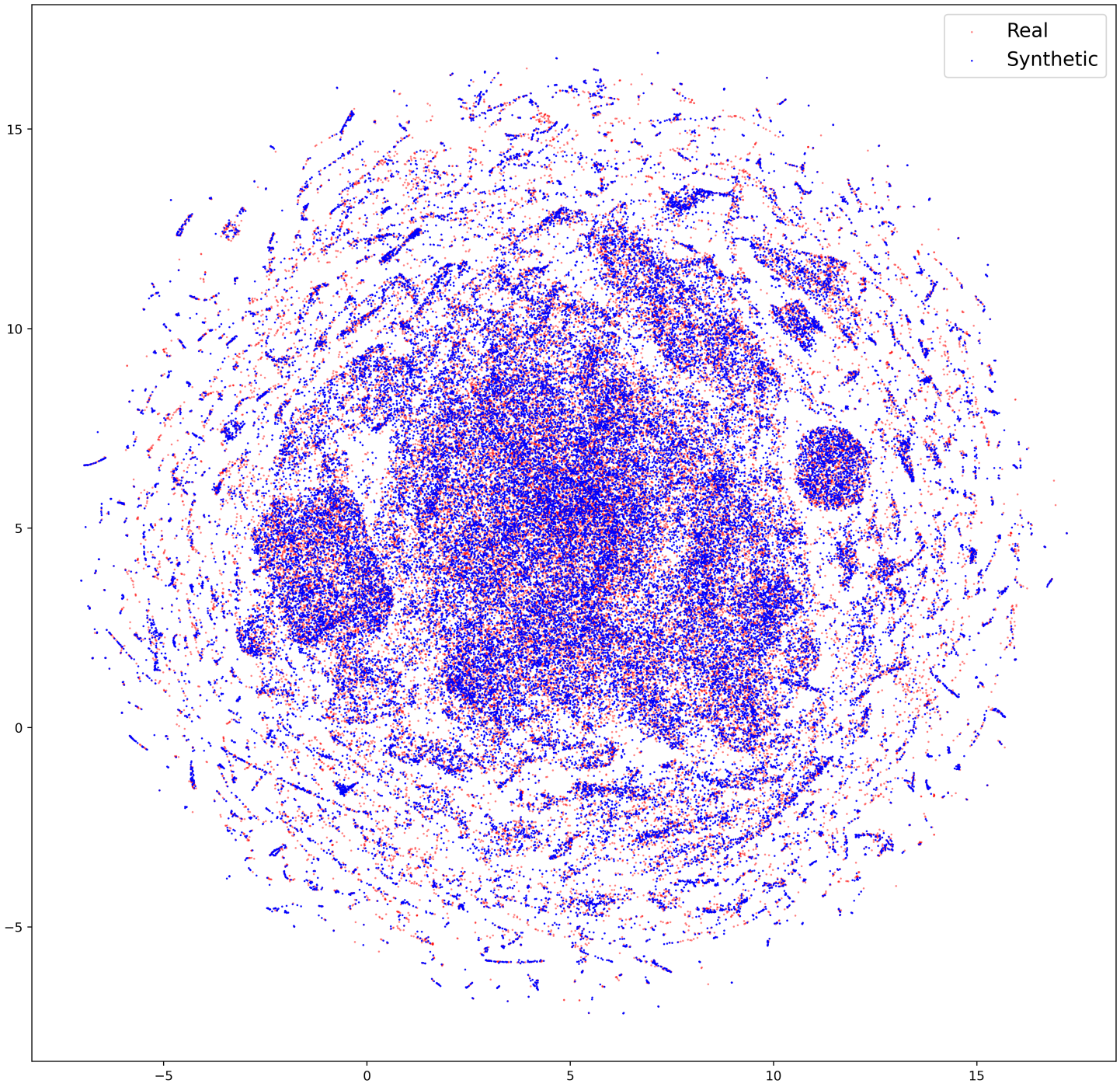

Figure 6.16: UMAP dimensionality reduction of the real and synthetic malicious TCP packet data, which maps multi-dimensional datasets to two dimensionless attributes to help visualize.

6.9 Discussion

The baseline results from TRTR give us our benchmark performance on what can be achieved by a classifier trained on the real data and is what TRTS, TSTS, and TSTR should be close to, to prove characteristic preservation. TRTR showed above 99% for accuracy and recall. It also showed perfect scores of 1.0 for precision and specificity. TRTS showed perfect 1.0 scores in all metrics. This is great because it shows that a classifier trained on the real data can perfectly classify all the synthetic data. However, I recognize that getting perfect scores in TRTS is unusual, considering it is better than TRTR. This can be explained by thinking about the data loss that occurs during the LSTM generator's learning process. When the generator learns the characteristics of the real dataset, it will not be able to learn them all, which is good because it allows some anonymization to occur. TRTR got 99% in accuracy and recall as well as 1.0 in both precision and specificity. This means that it only misclassified a few samples, which in this case is three misclassifications which can be seen in Fig. 6.10. When it learned the characteristics of the real data, most likely whatever characteristics these packets contained that made them be misclassified was not learned. Therefore when the synthetic dataset was created, those characteristics were not conserved, allowing TRTS to get perfect 1.0 scores in all metrics.

TRTS scores being very close to that TRTR show that the LSTMs that created the data were able to capture the structure of a TCP packet accurately and what characteristics are a part of it. The slightly higher score is not much, only being 0.0001 higher, so I would consider them to have basically the same performance. This result is good, but TRTS also would not show the situation where there was mode collapse in the model. This is because if the LSTM models that created the synthetic data only created one benign or malicious packet repeatedly that the classifiers could correctly classify, perfect 1.0 scores would be seen. Therefore, the next two tests are performed to remove the possibility of mode collapse.

The test results for the classifier trained on only synthetic data had two tests performed on it: TSTS and TSTR. TSTS had a perfect 1.0 score in recall and above 0.9999 in accuracy,

precision, and specificity. This means that the classifier trained on synthetic data can differentiate between malicious and benign packets with almost perfect scores in all metrics. In other words, the LSTM models that created the synthetic data were able to accurately capture what makes a malicious packet malicious and what makes a benign packet benign. However, again this would not show mode collapse in the model, and this is where TSTR comes in. TSTR is a classifier that is only trained on synthetic data and is asked to classify the entire real dataset of 1,131,103 real malicious packets and 1,131,103 real benign packets. It can do this with above 99% in all metrics, which is an excellent result and is almost identical to TRTR. If there were any mode collapse in the model that created the synthetic data, the classifier trained on synthetic data would not be able to classify large portions of the real dataset. This is because it would not have been able to recreate packets that were characteristically similar to all packets in the real dataset, and the classifier would not have been able to learn how to classify them. The metrics of TSTR being very close to TRTR proves there is no mode collapse in the model. This, along with TRTS and TSTS, proves that the characteristics of the original dataset are captured in the synthetic dataset.

The Levenshtein ratio tests showed a mean of 86.95% similarity for the benign TCP packets and a mean of 87.5% similarity for the malicious TCP packets. Both results are very similar however the Levenshtein graphs in Fig. 6.8 and Fig. 6.6 look very different. The graph of the benign Levenshtein ratios is bimodal, whereas the malicious graph is unimodal. One explanation for this is that one of the modes in the benign graph is the packets being received at the capture server from outside the victim network traveling to the victim computers. The other mode would be packets received by the capture server coming from the computers inside the victim network and traveling outside the network. This would be further enforced by the fact that almost all the traffic in the malicious dataset originated from the attacking network, which made the Levenshtein graph unimodal.

In terms of anonymization performance, 86.95% similarity for the benign TCP packets and 87.5% similarity for the malicious TCP packets is a good result. The training data I had

was only for a small artificial network, so its range of IP addresses and other information is limited. This leads to lower variance in the training set and, consequently, higher similarity in the synthetic dataset. When using this method on a dataset from a large corporate network, I would expect much better anonymization results due to the higher variance in a training set from the more complex network architecture.

Another thing to consider is that neither the synthetic benign nor malicious datasets had any exact copies of data from the original dataset, which is very good. This shows that the LSTMs that created the synthetic dataset did not overfit the training data. On a final note, the payload from the original TCP packets was removed, which is where a lot of confidential information is kept. This acts as a further form of anonymization, and it was shown that this did not affect classification performance. This, coupled with the Levenshtein ratio results, leads me to conclude that the synthetic dataset has been sufficiently anonymized.

Finally, we get to the results of the UMAP dimensional reduction. In Fig. 6.15, you can see the UMAP projection of the real benign TCP packets in red and the synthetic benign TCP packets in blue. Here it can be seen that there are many small darker modes where clusters of data points have formed. What these modes are is not important; what is important is that these modes have been picked up by the LSTMs that created the synthetic dataset. This shows that LSTM models that created the synthetic dataset can generate a synthetic dataset that conserves the global structure and characteristics of the real data, which further reinforces the findings from the classification tests. Outside these modes, where more scattered data points can be seen, there is some overlap between real and synthetic, but there is also many synthetic points that are distinct. This more scattered area shows that the synthetic dataset is also different from the original data showing some anonymization and reinforcing findings from the Levenshtein anonymization test.

In Fig. 6.16, you can see the UMAP projection of the real malicious TCP packets in red and the synthetic malicious TCP packets in blue. Here the modes are less distinct than in the benign graph; however, they are still there. You can see a very large cluster in the center and

some larger clusters forming around the outskirts of the graph. These are, again, the modes of the malicious data and what they are is not important; what is important is that the synthetic data follows a similar pattern to the real data. This shows that the LSTM models that created the synthetic malicious dataset were able to accurately capture the modes and characteristics of the real data in the synthetic dataset, further reinforcing the findings of the classification test. Outside these modes, you can again see some distinct synthetic data points that do not overlap the real data points. However, in this graph, they are a little more tightly packed than in the benign graph, which would account for the slightly higher Levenshtein similarity ratio for the malicious data set. This still shows some level of anonymization and further reinforces the findings from the Levenshtein test.

Chapter 7

Conclusion and Future Work

The move from rule-based cybersecurity systems to machine learning-based cybersecurity systems meant better security but also that vast amounts of application-specific training data would be needed. This created a situation where there is lots of cybersecurity data out there, but almost all of it is held by private corporations. These corporations either do not want to or can not share it due to privacy concerns about what is contained within the network data. The data that is shared is usually heavily anonymized using traditional techniques such as stripping it of all IP addresses and other information or is only available in traffic flow form. These methods often strip the network data of attributes that would be useful for training machine learning algorithms.

Consequently, this thesis proposes using a character-level LSTM model to learn the characteristics of a dataset; then generate a new, anonymized, synthetic dataset with similar characteristics to the original. To that end, the goal of my research was to see if it was possible to anonymize a packet-level TCP network intrusion detection dataset using character-level LSTM models. In this thesis, there were three main parts (Chapters 4-6) that each built upon each other to reach the end goal.

In Chapter 4, I started with anonymizing a URL dataset that contained both malicious and benign URLs. This seemed like a logical starting point as URLs have a simple structure.

In this chapter, we saw that a character-level LSTM model was able to successfully learn the characteristics of what makes a malicious URL, malicious, and a benign URL, benign. This model then created a new, synthetic URL dataset that, due to information loss during the generation process, was anonymized. Once this was confirmed possible, the next logical step was to create DNS packets which are slightly more complicated as not only do they contain a URL but also packet header information as well.

In Chapter 5, the generation of a DNS packet dataset was attempted. Unfortunately, the dataset used was one I collected from my home network. This made it suffer from very low variability in the dataset which created poor anonymization results. It also had no malicious component to the dataset, so testing was more difficult. After confirming that the generation of syntactically correct DNS packets was possible, I decided to focus on tuning the architecture of the LSTM model by testing varying numbers of neurons. The results showed that 512 or 1024 neurons would work and showed similar performance. However, the 1024 neuron model took much longer to train, so the more efficient model was the one with 512 neurons. Once it was confirmed that DNS packet generation was possible, it was time to move on to the more difficult and ultimate goal of anonymizing a packet-level TCP dataset.

In Chapter 6, two modified versions of the LSTM used in Chapters 4 and 5 were used. They first learned the characteristics of the original packet-level TCP intrusion detection dataset. Then generated, a synthetic version of the dataset that was anonymized through the loss of information during the generation process. After testing, it was shown that the dataset was characteristically similar to the original while still being different enough to be considered anonymized. This anonymized dataset also only contained 100,000 samples, whereas the LSTM that generated it was trained on over 10 million. As an unintended consequence, the dataset was not only anonymized but also compressed. As we saw during the TSTR classification test, the classifier trained on the 100,000 synthetic samples was able to classify the real dataset almost perfectly. Although training set compression was not explored in this thesis, the initial results would seem to show success. However, I would leave it to future work to

thoroughly validate this claim.

As far as I know, this is the first attempt to use deep learning to anonymize network datasets, so no direct comparison can be made to other techniques. However, I do think this improves upon previous rule-based anonymization schemes. When using a rule-based method, a human decides what information to keep in the anonymized data. While this gives more control, it strips the data of features that may be useful to a deep learning model. In contrast, my method allows a deep learning model to decide what patterns and characteristics in the data are important to it, which it will replicate in the anonymized dataset. This, unfortunately, does give less control over how the anonymization is performed. However, in future work, I think it is possible to implement finer-grained control over not only what is anonymized but how much anonymization is performed.

All things considered, I would say this was a success. However, there is still room for improvement. For starters, the inclusion of the TCP payload would have been better, but it was removed in order to simplify the problem. Secondly, while the TCP dataset I used for training was the best at the time, a better approach would be to combine multiple IDS datasets into one. This would have given more variability in the training set and probably led to better anonymization results. Lastly, evaluating this method on an actual dataset from a large corporate network would have been ideal. Unfortunately, I was unable to get access to any. So, the approximation using the smaller synthetic network was the best I could do.

Despite all this, the experimental results suggest that I successfully used a character-level LSTM model to learn the characteristics of a dataset; then generate a new, anonymized, synthetic dataset with similar characteristics to the original. This means that a company could use this method to create an anonymized synthetic version of their network data. Then give this data to another company to develop a machine learning-based IDS for them without being concerned about leaking confidential information. The company developing the IDS can also be confident that when it is deployed on the customer's network, the performance should be almost the same; due to the characteristic similarity of the synthetic data. It is my hope that this

research also paves the way for the publication of private network datasets, giving academics better access to high-quality, real-world data for use in their research.

Bibliography

[1] Alankrita Aggarwal, Mamta Mittal, and Gopi Battineni. Generative adversarial network: An overview of theory and applications. *International Journal of Information Management Data Insights*, page 100004, 2021.

[2] Thomas M. Chen and Patrick J. Walsh. Chapter 3 - guarding against network intrusions. In John R. Vacca, editor, *Network and System Security (Second Edition)*, pages 57–82. Syngress, Boston, second edition edition, 2014.

[3] Adriel Cheng. Pac-gan: Packet generation of network traffic using generative adversarial networks. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0728–0734. IEEE, 2019.

[4] Antonia Creswell, Tom White, Vincent Dumoulin, Kai Arulkumaran, Biswa Sengupta, and Anil A Bharath. Generative adversarial networks: An overview. *IEEE Signal Processing Magazine*, 35(1):53–65, 2018.

[5] Mostafa Dehghani, Hosein Azarbonyad, Jaap Kamps, and Maarten de Rijke. Share your model instead of your data: Privacy preserving mimic learning for ranking, 2017.

[6] Cynthia Dwork. Differential privacy: A survey of results. In Manindra Agrawal, Dingzhu Du, Zhenhua Duan, and Angsheng Li, editors, *Theory and Applications of Models of Computation*, pages 1–19, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Inc Free Software Foundation. Wireshark. `www.wireshark.org`, 2022.

[8] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, 11 1997.

[9] Harjinder Kaur, Gurpreet Singh, and Jaspreet Minhas. A review of machine learning based anomaly detection techniques. *arXiv preprint arXiv:1307.7286*, 2013.

[10] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2014.

[11] Lawrence Berkeley National Laboratory. tcpmkpub. `https://www.icir.org/enterprise-tracing/tcpmkpub.html`, 2013.

[12] Carlos Lara. Character-level lstm in pytorch. `https://github.com/LeanManager/NLP-PyTorch/blob/master/Character-Level%20LSTM%20with%20PyTorch.ipynb`, 2019.

[13] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion detection system: A comprehensive review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.

[14] Samaneh Mahdavifar and Ali A Ghorbani. Application of deep learning to cybersecurity: A survey. *Neurocomputing*, 347:149–176, 2019.

[15] Mohammad Saiful Islam Mamun, Mohammad Ahmad Rathore, Arash Habibi Lashkari, Natalia Stakhanova, and Ali A Ghorbani. Url dataset (iscx-url2016). `https://www.unb.ca/cic/datasets/url-2016.html`, 2016.

[16] Sándor Molnár, Péter Megyesi, and Géza Szabó. How to validate traffic generators? In *2013 IEEE International Conference on Communications Workshops (ICC)*, pages 1340–1344. IEEE, 2013.

[17] Gonzalo Navarro. A guided tour to approximate string matching. *ACM computing surveys (CSUR)*, 33(1):31–88, 2001.

[18] NETRESEC. Splitcap. www.netresec.com/?page=SplitCap, 2021.

[19] Ruoming Pang, Mark Allman, Vern Paxson, and Jason Lee. The devil and packet trace anonymization. 36(1):29–38, jan 2006.

[20] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. A survey of network-based intrusion detection data sets. *Computers & Security*, 86:147–167, 2019.

[21] J. Rosenberg. Chapter e6 - embedded security. In Augusto Vega, Pradip Bose, and Alper Buyuktosunoglu, editors, *Rugged Embedded Systems*, pages e1–e74. Morgan Kaufmann, Boston, 2017.

[22] Ahmed Shafee, Mohamed Baza, Douglas A. Talbert, Mostafa M. Fouda, Mahmoud Nabil, and Mohamed Mahmoud. Mimic learning to generate a shareable network intrusion detection model. In *2020 IEEE 17th Annual Consumer Communications & Networking Conference (CCNC)*, pages 1–6, 2020.

[23] Iman Sharafaldin, Amirhossein Gharib, Arash Habibi Lashkari, and Ali A. Ghorbani. Towards a reliable intrusion detection benchmark dataset. 2017.

[24] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *ICISSP*, 2018.

[25] Harsh Trivedi. Minimal tutorial on packing and unpacking sequences in pytorch. https://github.com/HarshTrivedi/packing-unpacking-pytorch-minimal-tutorial, 2019.

[26] Laurens Van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[27] G. Van Houdt, C. Mosquera, and G Nápoles. A review on the long short-term memory model. *Artificial Intelligence Review*, 53:5929–5955, 2020.

[28] Spencer Vecile, Kyle Lacroix, Katarina Grolinger, and Jagath Samarabandu. Malicious and benign url dataset generation using character-level lstm models. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, 2022. (In Print).

[29] K. Weiss, T.M. Khoshgoftaar, and D. Wang. A survey of transfer learning. *Journal of Big Data*, 3:9, 2016.

Appendix A

Detailed Preprocessing Information

A.1 Monday

Shrinking Monday using Splitcap did not help since almost all the data was TCP. So to filter out all the protocols except TCP, the filter used in Wireshark was “tcp && !(http || tls || icmp || nbss || smb || ftp || kerberos || dns || ftp-data || ldap || ssh || websocket || opcua)”. After that filter was applied, the remaining packets were all TCP. The next step included a lot of trial and error to get Wireshark not to crash when exporting the packet dissections, but eventually, it worked. This produced a massive 250-gigabyte JSON file that gets its size from including all the different packet attributes in byte stream format. Most of these attributes are unnecessary as we only need the packet as a single hexadecimal string. Unfortunately, I could not find any other way to get the packet exported as a hexadecimal string.

The next step was to extract only the byte streams from this JSON file, label them benign and save them in CSV format for ease of use with machine learning libraries. However, the massive size of the JSON file made it impossible to load it into Python using the traditional JSON library since it would multiply in size even more and crash the system. So a package called ijson was used, which loads the JSON file as a stream instead of the entire thing at once, allowing it to be iterated through without using vast amounts of random access memory

(RAM). Once that was figured out, the byte streams were extracted using a loop and saved in CSV format with the benign label leaving a total of 9,156,151 benign packets.

A.2 Tuesday

Tuesday's original file size was about 10.8 gigabytes, containing about 11.5 million packets which are a mix of benign and malicious packets. Since Monday has more than enough benign traffic, we only need to extract the malicious packets from Tuesday's file. In Section 6.2.1 we saw that there were only about 110,000 malicious packets in Tuesday's PCAP file and that they all came from the IP 172.16.0.1. Exploiting this information, Splitcap was used and split the original PCAP file by only including packets originating from 172.16.0.1 in the new smaller file. Shrinking Tuesday's PCAP file size from 10.8 gigabytes to 42 megabytes which were easily loaded into Wireshark. From there only packets originating from 172.16.0.1 were kept and all other protocols except TCP were filtered out using the Wireshark filter "ip.src==172.16.0.1 && tcp && !(http || tls || icmp || nbss || smb || ftp || kerberos || dns || ftp-data || ldap || ssh || websocket || opcua)". The remaining data's packet dissections were exported, creating a JSON file of about 670 megabytes. Next, the packet streams will need to be extracted from the JSON file and mapped to the labeled TCP flows contained in the original CSV files that came with the dataset.

First, the JSON file was imported, and the UID information with its corresponding packet was extracted and stored in a list. This resulted in 36,178 packets being extracted from the JSON file. Next, the CSV file was imported using pandas, and all rows (flows) except the ones containing the malicious IP 172.16.0.1 were dropped. This resulted in 13,835 labeled flows, and when the UIDs of these rows were checked for duplicates, it showed that there were 196 duplicate UIDs. Because of these duplicates, I could not just drop all the benign labeled rows. Some duplicated UIDs may have malicious and benign traffic flows associated with them. So in the case of multiple labels, they will need to be dropped as their correct label can not be

ascertained. If only the malicious rows were kept, we would not see these contradicting labels and may accidentally include benign packets in the final dataset. The UID of the flows will next need to be compared to the UID information of individual packets from the JSON file to map the flow labels to the individual packets that were part of that flow.

Next, a simple nested for loop was used that compared each flow (outer loop) to all the packets (inner loop) to find the packets with a UID that matched the UID tuple of a flow. Once matches were found, the label(s) were stored next to that packet in a list. Since all the information was in Pandas DataFrames, I initially used the "iterrows" function to iterate. However, this was extremely slow, only running at a rate of about one iteration per second, which means that this process would take about 4 hours to complete, and Tuesday had the smallest amount of data by far, so it was only going to get worse. After some research, I found that transforming the Pandas data frames into Numpy arrays would significantly speed up the iteration process. It increased the iteration speed to 40 iterations per second, dropping the time from almost 4 hours to about 6 minutes. After this, there were 72 packets with no label, 14 with conflicting labels, and 1621 labeled benign. This resulted in 1707 packets being dropped, leaving 34,471 malicious packets, which were saved with their resulting label in CSV format.

A.3 Wednesday

Wednesday's original file size was about 13 gigabytes, including about 13.7 million packets, which are a mix of benign and malicious packets; similarly to Tuesday, we only need the malicious packets. In Section 6.2.1, it was shown that there are almost 1.4 million malicious packets, all originating from the IP 172.16.0.1. So again, we will exploit this using Splitcap to reduce the file size by only keeping packets with the IP 172.16.0.1. This shrunk the file size from 13 gigabytes to 2.3 gigabytes which was much more manageable in Wireshark. The remaining packets were filtered using the filter "ip.src==172.16.0.1 && tcp && !(http || tls || icmp || nbss || smb || ftp || kerberos || dns || ftp-data || ldap || ssh || websocket || opcua)" which

kept only TCP and packets originating from 172.16.0.1. After this, the packet dissections were exported, resulting in a JSON file of almost 23 gigabytes. Next, the packet streams need to be extracted from the JSON file and mapped to the labeled TCP flows contained in the original CSV files that came with the dataset.

The JSON file was imported, and the packets and their UID's extracted. This resulted in 1,264,816 packets being extracted from the JSON file. Next, the CSV file was imported using pandas, and all rows (flows) except the ones containing the malicious IP address 172.16.0.1 were dropped. This resulted in 254,821 labeled flows, and when the UIDs of these rows were checked for duplicates, it showed that there were 16,268 duplicate UIDs. Next, the UIDs of the flows will need to be compared to the UID information of individual packets that were extracted from the JSON file. This will allow us to map the flow labels to the individual packets that were part of that flow. It will also allow us to remove any benign packets or packets with conflicting labels.

The same nested loop structure with Numpy arrays that was used on Tuesday was attempted, but this was super slow and estimated 85 hours to complete. So after some research, a further optimization was found. In some situations, the query function from the Pandas library can be faster than just iterating over each row to find a UID match. So instead of the outer loop being the flows and the inner loop being the packets. The outer loop was the packets, and the inner loop was gone and replaced with a query that searched for which flows matched the UID tuple of each packet. If there were multiple matches, each label was stored in a list next to the corresponding packet. This optimization resulted in the time going from 85 hours to just 18 hours which is much more manageable. Once completed, there were 2,829 packets with no label, 977,702 packets with conflicting labels, and 16,862 benign packets. After dropping the benign packets, packets with no labels, and packets with conflicting labels, we are left with 267,423 malicious packets. A large number of packets were dropped; however, this is not necessarily bad since all of them were DoS packets, which are highly repetitive. If all 1.2 million of them were kept, it could have caused the LSTM to overfit on them and experience mode

collapse. This is because it would be more than all the other attack types combined, creating a very unbalanced dataset.

A.4 Thursday

Thursday had a smaller file size, only about 8 gigabytes. This includes about 9.2 million packets which are a mix of benign and malicious packets, and similarly to Tuesday, we only need the malicious packets. In Section 6.2.1 we saw that there are almost 54,000 malicious packets, except this time they come from two addresses, 172.16.0.1 and 192.168.10.8. Fortunately, the 8-gigabyte file was not too much for Wireshark, and no Splitcap step was needed. Once loaded into Wireshark the packets were filtered using the filter "ip.src==172.16.0.1 && tcp && !(http || tls || icmp || nbss || smb || ftp || kerberos || dns || ftp-data || ldap || ssh || websocket || opcua)" and the packet dissections were exported. This resulted in a JSON file that was about 6.3 gigabytes in size. Then, the JSON file was imported into Python, and the packet streams and their UIDs were extracted. This resulted in 371,673 packets being extracted from the JSON file. Finally, the extracted packet streams will need to be mapped to the labeled TCP flows contained in the original CSV files that came with the dataset.

To do this, the CSV file was first imported using pandas, and all rows (flows) except the ones containing the malicious IP addresses 172.16.0.1 and 192.168.10.8 were dropped. This resulted in 3,297 labeled flows, and when the UIDs of these rows were checked for duplicates, it showed that there were 259 duplicate UIDs. The following steps were the same as the previous days in terms of comparing the UIDs of the flows and the packets to map the packets to their corresponding flow and label. However, something interesting was noticed. When the loop method that involved using the query like in Wednesday was attempted, it showed that the process would take about 3 hours and 20 minutes. This seemed too long considering how small this data set was compared to Wednesday. So the original nested loop structure from Tuesday was attempted and finished in only 14 minutes. This showed that the query

structure of comparison is only faster when the dataset is enormous, and when the dataset is smaller, a nested loop structure using Numpy arrays is much faster. Once the mapping was complete, there were 323,302 packets with no label, 478 packets with conflicting labels, and 1,311 benign packets. After dropping the benign packets, packets with no labels, and packets with conflicting labels, we are left with 46,582 malicious packets.

A.5 Friday

Like Thursday, Friday's pcap file was smaller, only being about 8.5 gigabytes in size. This included about 9.9 million packets which were a mix of benign and malicious packets, so again the malicious ones needed to be extracted. In Section 6.2.1 we saw that there are about 740,000 malicious packets, except this time, due to the botnet traffic it comes from many IP addresses. These addresses are 192.168.10.12. 192.168.10.14, 192.168.10.15, 192.168.10.17, 192.168.10.5, 192.168.10.8, 192.168.10.9, 205.174.165.73 and 172.16.0.1. To extract all the TCP traffic coming from these addresses the Wireshark filter used was "(ip.src==192.168.10.12 || ip.src==192.168.10.14 || ip.src==192.168.10.15 || ip.src==192.168.10.17 || ip.src==192.168.10.5 || ip.src==192.168.10.8 || ip.src==192.168.10.9 || ip.src==205.174.165.73 || ip.src==172.16.0.1) && tcp && !(http || tls || icmp || nbss || smb || ftp || kerberos || dns || ftp-data || ldap || ssh || web-socket || opcua)". After the filter was applied, the packet dissections were exported, resulting in a JSON file about 55 gigabytes in size. The large size of the JSON file meant it needed to be imported in the same way as Monday's file using ijson. Using a loop, the UID tuple of each packet, along with the byte stream, was extracted and stored in a Pandas DataFrame. This resulted in 3,348,380 packets, and since there were only about 740,000 malicious packets counted in the flow CSV file, this means there are many benign packets mixed in there. Next, the extracted packet streams will need to be mapped to the labeled TCP flows contained in the original CSV files that came with the dataset.

To do this, the CSV file was first imported using pandas, and all rows (flows) except the ones

containing one of the malicious IPs mentioned above were dropped. This resulted in 288,923 labeled flows, and when the UIDs of these rows were checked for duplicates, it showed that there were 6,126 duplicate UIDs. Next, the UID of the flows will be compared to the UID information of individual packets from the JSON file to map the flow labels to the individual packets that were part of that flow. A nested loop structure of comparison was very inefficient due to the large size of the arrays, so the query method will be used. This process took 45 hours to complete, and when it was done, there were 2,509,620 packets with no label, 6,126 packets with conflicting labels, and 0 benign packets. After dropping the packets with no labels and conflicting labels, we are left with 832,634 malicious packets. This is interesting because only about 740,000 malicious packets were counted in the flow CSV file, so there are about 90,000 extra packets labeled. This was either caused by a problem with the labeling system or an incorrect original count in the supplied CSV.

Appendix B

Hardware Environment

CPU

- AMD Ryzen 9 3900X
- Base Clock 3.8 GHz, Boost Clock 4.1 GHz
- 12 Cores, 24 Threads

GPU

- Asus ROG Strix Nvidia GeForce RTX 3090
- 24GB GDDR6X VRAM
- 1890 MHz Clock
- 10496 CUDA Cores

RAM Memory

- Corsair Vengeance DDR4 3600 Mhz C18
- 96GB (2x32GB, 2x16GB)

SSD

- Samsung 970 Evo Plus
- 1TB

Curriculum Vitae

Name:	Spencer Vecile
Post-Secondary Education and Degrees:	University of Western Ontario London, ON 2015 - 2021 B.E.Sc, SE
Honours and Awards:	The Western Scholarship of Excellence 2015
	Dean's Honour List 2019, 2020
	Edward and Janet Schroeder 125th Anniversary Alumni Awards 2020
	University of Western Ontario Undergraduate Summer Research Award 2020
Related Work Experience:	Undergraduate Student Research Internship The University of Western Ontario 2020

Publications:

Spencer Vecile, Kyle Lacroix, Katarina Grolinger, and Jagath Samarabandu. Malicious and benign url dataset generation using character-level lstm models. In *2022 IEEE Conference on Dependable and Secure Computing (DSC)*, 2022. (In Print).