Using Autoencoder Feature Residuals to Improve Network Intrusion Detection

by

Brian Lewandowski

A Dissertation

Submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

In partial fulfillment of the requirements for the

Degree of Doctor of Philosophy

in

Computer Science

August 1, 2023

APPROVED:

Professor Randy C. Paffenroth Worcester Polytechnic Institute Advisor	Professor Fabricio Murai Worcester Polytechnic Institute Committee Member
Professor Yanhua Li Worcester Polytechnic Institute Committee Member	Dr. Nandi Leslie Raytheon Technologies External Committee Member
Craig A. Shue, Ph.D. Worcester Polytechnic Institute Department Head Computer Science Department	

Abstract

Network intrusion detection is a constantly evolving field with researchers and practitioners constantly working to keep up with novel attacks and growing amounts of network data. Traditionally, signature-based techniques and deep packet inspection have been employed, however, the volume of data and complexity of network attacks has made these techniques overbearing and susceptible to zero-day attacks. For this reason, there has been a shift in focus to explore the power of deep learning anomaly-based methods to perform network intrusion detection.

In this dissertation we develop and explore several deep learning techniques and their application to performing anomaly-based network intrusion detection. Central to the work is an exploration of the development of unique feature sets using autoencoder feature residuals which have traditionally been overlooked in favor of aggregate residuals. We show that using feature sets generated using autoencoder feature residuals provide an improvement in downstream classifier performance compared to an original feature set for network intrusion detection. In doing so, we find that these overlooked byproducts of anomaly-based methods can be used as a drop-in replacement for an original feature set, meeting or exceeding its performance.

Acknowledgements

I would first like to thank and acknowledge my advisor, Randy Paffenroth, of Worcester Polytechnic Institute. He agreed to take me on as a student during the pandemic despite not being able to meet in person at the time. Looking back, I don't recall a time that wasn't filled with positivity and encouragement during our work together. I am thankful for his emphasis on rigor and mathematics that has done nothing but benefit myself and this work.

Additionally, I would like to thank my committee members from Worcester Polytechnic Institute, Yanhua Li and Fabricio Murai. They have both provided excellent encouragement and insight into the research that has helped shape this work.

I would like to thank my employer, Raytheon Technologies, for providing me with the flexibility and funding to pursue this work. In particular, I would like to thank Nandi Leslie, who volunteered her time to be a member of my dissertation committee. I am grateful for her insights into my research. Thank you to all of my co-workers for their encouragement and positivity throughout these years.

I would like to thank my parents, Margaret and Walter, for all of their love and instilling me with a strong work ethic and an emphasis on learning. Thank you to my sisters, Tisia and Ashley, Nana, and extended family of aunts, uncles, and cousins who have provided me with so much joy throughout my life and have been so encouraging since I started this work.

Most of all I would like to thank my wife Ann for her endless love and support and my children, Annie and Brian, for the joy and happiness they bring to my life every day. I would not have been able to do this without them. Finally, I would like to thank Millie for giving me a reason to get up every morning... literally.

Papers Contributing to this Dissertation

Accepted Papers

N. Bahadur, **B. Lewandowski**, and R. Paffenroth. (2022). "Dimension Estimation Using Autoencoders and Application," In: M.A. Wani, B. Raj, F. Luo, D. Dou (eds) Deep Learning Applications, Volume 3. Advances in Intelligent Systems and Computing, vol 1395.

B. Lewandowski, R. Paffenroth and K. Campbell, "Improving Network Intrusion Detection Using Autoencoder Feature Residuals," 2022 4th International Conference on Data Intelligence and Security (ICDIS), 2022, pp. 31-39.

B. Lewandowski and R. Paffenroth, "Autoencoder Feature Residuals for Network Intrusion Detection: Unsupervised Pre-training for Improved Performance," 2022 21st IEEE International Conference on Machine Learning and Applications (ICMLA) Special Session on Cybersecurity and Big Data, 2022, pp. 1334-1341.

B. Lewandowski, "Guidelines and a Framework to Improve the Delivery of Network Intrusion Detection Datasets", Accepted to the 20th International Conference on Security and Cryptography (SECRYPT).

B. Lewandowski and R. Paffenroth, "One-class Classification Using Autoencoder Feature Residuals for Improved IoT Network Intrusion Detection", Accepted to the 13th International Workshop on Security, Privacy, and Trust for IoT (IoTSPT).

Accepted Papers (Continued)

B. Lewandowski and R. Paffenroth, "Autoencoder Feature Residuals for Network Intrusion Detection: One-class Pre-training for Improved Performance", Submitted to the Journal of Machine Learning and Knowledge Extraction (MAKE) Special Issue on Deep Learning and Applications.

Additional Papers

B. Lewandowski and R. Paffenroth, "Analyzing the Effectiveness of Autoencoder Feature Residuals Using Time Series Data for Network Intrusion Detection", To be submitted to the 22nd IEEE International Conference on Machine Learning and Applications (ICMLA) Special Session on Cybersecurity and Big Data.

Funding and Technical Acknowledgements

The results of this dissertation were made possible through the Raytheon Technologies AIMLeaders program which provided funding to perform this research.

A majority of the results reported in this dissertation were obtained using a high-performance computing system, the Turing Research Cluster, provided to WPI through the NSF grant DMS-1337943.

This document does not contain technology or Technical Data controlled under either the U.S. International Traffic Arms Regulations or the U.S. Export Administration Regulations.

Executive Summary

Overview of Our Work

In this dissertation we aim to balance both theoretical and practical applications of autoencoders in the field of network intrusion detection. We focus on several techniques with the majority of the focus centered on using autoencoder feature residuals as features to perform classification of network attacks.

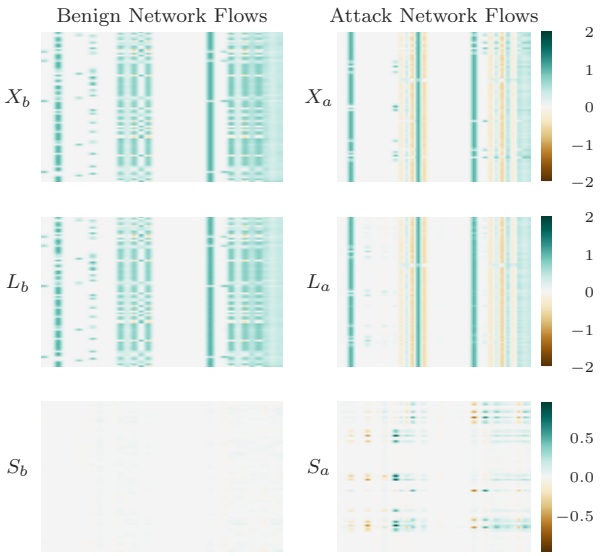


Figure 1: A comparison of an original set of features X, the autoencoder reconstruction L, and the autoencoder feature residuals S, using 100 samples of network flow data. One can see that the autoencoder feature residuals for benign samples, S_b, on the bottom left side of the plot, is sparse and significantly lower in magnitude than the autoencoder feature residuals for attack samples, S_a, on the bottom right-hand side. Differences such as these, in the autoencoder feature residuals between benign and attack samples, allow our technique to improve downstream classifier performance.

We first highlight the application of autoencoder dimension estimation to detect large scale network attacks in Chapter 4. In doing so, we find that the technique works well for attacks such as a distributed denial of service, which have network-wide impacts. On the other hand, the technique was not designed for

Table 1: Samples of some of our best results performing network intrusion detection using autoencoder feature residuals in combination with a variety of downstream classifiers. Values in green highlight the improved scores.

Classifier	**Dataset**	**Original F1-Score**	**Improved F1-Score**
KNN	CTU13 Scenario 6	0.800	0.870
LR	CICIDS2017 Wednesday	0.900	0.940
MLP	NF-ToN-IoT-V2	0.973	0.975
MLP	NF-UNSW-NB15-V2	0.780	0.825
IF	ToN-IoT	0.393	0.770
OCSVM	BoT-IoT	0.793	0.934
LOF	ToN-IoT	0.758	0.936

detecting more subtle attacks that we must concern ourselves with in the area of network intrusion detection.

We then move on to develop our technique for generating and using feature sets that take advantage of autoencoder feature residuals in Chapter 5. After outlining the general technique we discuss the properties of autoencoder feature residuals that provide the opportunity to generate optimal features for downstream tasks. Along with this, we contrast these properties with conditions found in data that may reduce the technique's effectiveness, and close with a comparison to using aggregate residuals for anomaly detection.

Having outlined the theory of using autoencoder feature residuals, we focus Chapter 6 on the application of this technique to network intrusion detection. Through exhaustive empirical studies we show that using autoencoder feature residuals as features for downstream classifiers improves performance over some initial set of features for network intrusion detection, and at a minimum does not reduce the comparative performance.

We close in Chapter 7 with a set of practical work that was formed through dealing with the challenging properties of network data. In this work, we develop a set of guidelines and open source framework to improve the development and hand off of network intrusion detection datasets between researchers.

Our Contributions

- We demonstrate how autoencoder dimension estimation can be applied effectively to network intrusion detection using network flow data from a network under attack conditions.
- We propose a novel method of using autoencoder feature residuals to improve classifier performance for network intrusion detection using network flow data. We develop its theory and define the key properties surrounding the technique.
- We develop the first application of using autoencoder feature residuals as input to classical machine learning algorithms, various neural network architectures, and several one-class classifiers for network intrusion detection problems.
- We provide a comprehensive analysis to show that usage of autoencoder feature residuals provides a general increase in performance compared to using a baseline set of original features for network intrusion detection. This analysis is performed using more than ten cybersecurity scenarios, eleven downstream classifiers, and two encodings of network intrusion detection data. Additionally, statistical support using the two-sample Kolmogorov-Smirnov test is provided for the analysis.
- We identify and demonstrate the potential data compression benefits that can be found when using autoencoder feature residuals.
- We identify a set of common limitations that affect the handoff of network intrusion detection datasets between researchers. In response to these lim-

itations we develop a set of guidelines for dataset researchers to follow to help mitigate these limitations. An open source containerized framework is developed which implements the guidelines such that it can be used by researchers.

- We have made the key portions of our work available for the public including implemented code and data such that it can benefit other researchers[1][2][3][4].

[1] https://github.com/WickedElm/feature_residuals
[2] https://github.com/WickedElm/feature_residuals_with_pretraining
[3] https://github.com/WickedElm/feature_residuals_for_iot
[4] https://github.com/WickedElm/niddff

Contents

1 Introduction

1.1 Motivation

We currently rely on networks for nearly all aspects of our lives today. As individuals, we often take for granted having connectivity to the internet for both work and leisure. Networks serve as a critical backbone to the efficiency and scale of operating modern businesses. The benefits of network technology and our reliance on it has caused networks to grow in both size and complexity in recent decades [51]. It is now common to use networks for transactions that require sensitive data such as banking, completing purchases with credit cards, storing competition sensitive trade data, and to support critical infrastructure [102, 57].

While a majority of network users remain respectful of sensitive data and use networks responsibly, there remain numerous bad actors that perform network attacks. The motivation for these attacks include a number of factors such as monetary gain, personal beliefs, as well as state-sponsored interests [55]. The number of these attacks increased 600% globally during the COVID-19 pandemic and is expected to cost $10.5 trillion globally by 2025 [96]. Our work is intended to help stem this staggering damage that network attacks impose on both organizations and individuals. To achieve this goal we explore the application of deep learning to network intrusion detection (NID) with a focus on techniques that detect network attacks by treating them as anomalies.

1.2 Overview

We begin in Chapter 2 by guiding the reader through the necessary preliminaries that support the development of our techniques with a focus on network intrusion detection and anomaly detection. We discuss the relevant machine and deep learning algorithms used along with our technique, however, a particular focus is spent on autoencoders as they are central to the remainder of the work presented.

Having covered the central concepts surrounding the background of our technique, we move into Chapter 3 where we review recent related works in the area of network intrusion detection. In addition, this chapter covers robust autoencoders, which provided the seeds for the exploration of autoencoder feature residuals.

Moving into our research we first cover the application of autoencoder dimension estimation to network intrusion detection in Chapter 4. In this chapter we review the technique and show that while it performs well on distributed denial of service (DDoS) attacks, it struggles to detect more subtle network attacks.

Chapter 5 is dedicated to the development of autoencoder feature residuals. Here, we formally define what we mean by autoencoder feature residuals and how to generate them using autoencoders. Additionally, we review the properties of autoencoders in relation to autoencoder feature residuals and define the various feature sets explored in our research based on these properties.

Having reviewed the general technique for using autoencoder feature residuals, we move into Chapter 6 to explore the particulars regarding applying the technique to network intrusion detection. Here we review the feature encodings used, model architectures explored, hyperparameter selection, as well as the results obtained detecting network attacks in conjunction with a wide range of classifiers and network intrusion detection scenarios. It is through these empirical results that we

show the utility of the technique in the area of network intrusion detection.

While working through the application of autoencoder feature residuals to network intrusion detection, we found a number of constructive complexities regarding the state of network intrusion detection datasets. We explore these shortcomings in Chapter 7 along with a set of guidelines and a framework we developed to help improve the situation for future researchers.

We conclude our discussion in Chapter 8 and outline what we believe to be fertile ground for future work extending this research in Chapter 9.

2 Background Information

2.1 Cyber Defense

Cyber defense is a broad area which can be considered the protection of any computer asset. One could break down the area of cyber defense into the types of attacks being defended against [28], however, for this work it is more suitable to break the area down into the main goals of defenders: network intrusion detection, host intrusion detection (HID), and malware detection/analysis.

2.1.1 Network Intrusion Detection

NID analyzes network data in order to detect network attacks. We refer to the system that performs NID as a network intrusion detection system (NIDS). The NIDS consists of a combination of software and network hardware to perform the required analysis. Depending on the network, the hardware can be embedded on existing network infrastructure devices or have dedicated hardware. Depicted in Figure 2 one can see an example of a network that utilizes a NIDS.

In general, there are two broad categories of methods that we consider when discussing NIDS: signature-based and anomaly-based [64]. With signature-based methods, we attempt to identify a network attack based on previously seen patterns, referred to as a signature, for a known attack [64]. Anomaly-based methods attempt to set a baseline for what is considered normal operations for a network and detect any deviations from this baseline [64]. While signature-based methods are effective against known attacks, they have the drawback of failing to detect novel, zero-day attacks, that the NIDS has not seen previously. Comparatively, anomaly-based methods are more effective at detecting zero-day attacks, however,

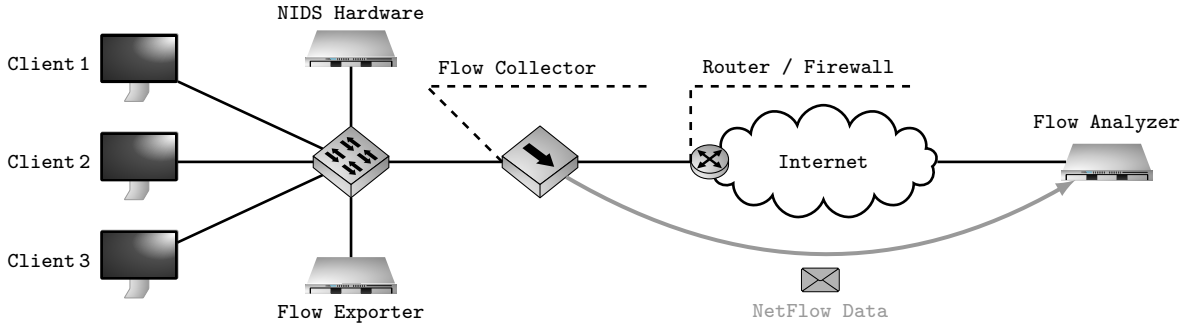


Figure 2: A simplified network which makes use of a network intrusion detection system with dedicated hardware and collects and analyzes network flow data. These pieces would be used in combination with the pictured firewall as well as host intrusion detection systems present on Clients one through three. Our work could be deployed to the NIDS hardware or a combination of the NIDS hardware as well as utilizing a separate training server such as the depicted flow analyzer hardware.

they often suffer from high false-alarm rates. Due to the significant threat that zero-day attacks pose, current research is focused toward anomaly-based methods or methods that consist of a hybrid of both signature and anomaly-based methods [64, 28, 10].

2.1.2 Network Data

In order to perform NID, a NIDS must analyze network data. This data comes in one of two formats referred to as packet capture (PCAP) and network flow (NetFlow) data. The source for both of these formats is binary, however, for our application they are generally extracted into a format suitable for machine learning such as comma-separated value (CSV) files.

PCAP data is used to gather a complete set of packets from a given network or relevant section of a network. This format for data is dense and consists of all of the contents of the captured network packets including payload data as shown in Figure 3. The main advantage of PCAP data is its completeness since no data

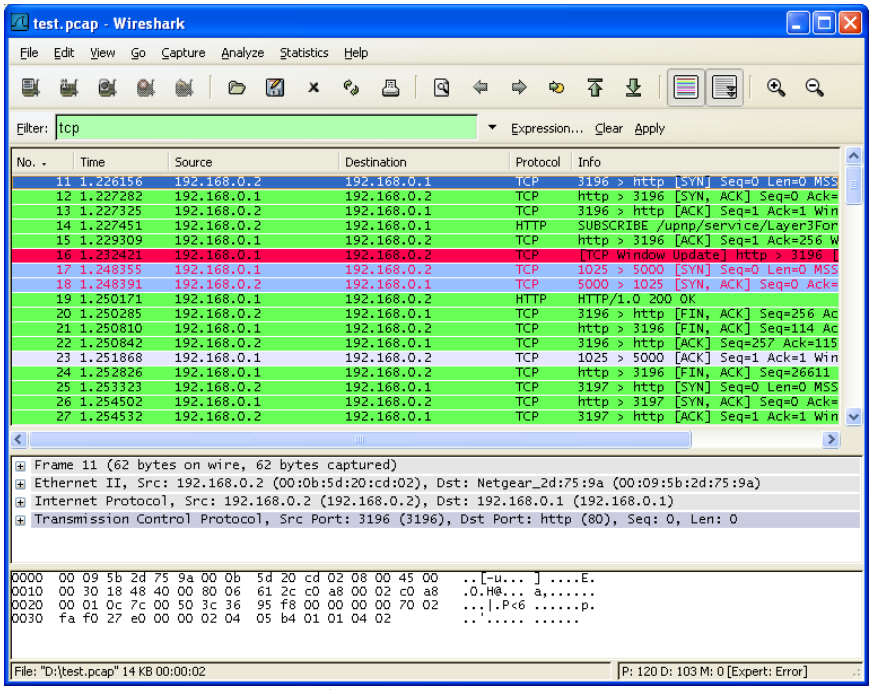


Figure 3: A sample extraction of packet capture (PCAP) data taken using Wireshark. This format is dense and includes all the information contained in the packets that were captured including all headers and the data payload.

is lost compared to what was communicated on a network. While this format is necessary to perform deep packet inspection, the main drawback of using it for machine learning is that the volume of data captured becomes overwhelming even for small networks.

As a way to mitigate the amount of data collected when capturing full packet data, Cisco Systems developed the original NetFlow standard [99]. While there have been multiple iterations since its first introduction, the main idea behind NetFlow data is that rather than capturing all network packet data, statistics are collected that describe network connections between hosts. These connections are generally described using the 5-tuple: (source IP, source port, destination IP, destination port, protocol). For a given connection, statistics such as number of bytes and packets transmitted are collected. It should be noted that there are multiple

standards used for NetFlow data and that the fields collected are dependent on both the choice of standard for a given network and how the network administrator has the network configured. For example, when using the IP Flow Information Export (IPFIX) Protocol an administrator can customize what network data to analyze [25].

There are two primary ways that network flow data is collected. For many research applications, cybersecurity scenarios are executed in a lab where source PCAP data is collected initially. After this has been completed, tools such as Zeek[5] are utilized to generate NetFlow data, which is delivered as a dataset for downstream researchers. In practice, however, there is generally dedicated network hardware and software that consists of a NetFlow exporter, collector, and analyzer as depicted in Figure 2. These resources are used to extract the NetFlow data into a format that is usable for machine learning. It is worth noting that the term NetFlow data is used commonly in literature to pertain to any data that uses the traditional 5-tuple in order to capture summary statistics about the connection.

2.1.3 Challenging Properties of Network Data

Having had a brief discussion on the format and type of data used for cyber defense, we will now discuss the key properties of the data that pose challenges when using it for machine learning. In the following discussion regarding the properties of network data we do not distinguish between PCAP, NetFlow data, or other variants specifically, as the key properties discussed manifest in all of these data formats.

[5]https://zeek.org/

Imbalanced Data

Network data used for cybersecurity suffers from being imbalanced when comparing benign network data and attack network data [12]. The large majority of network traffic will be benign for a given network compared to any attack network traffic. One example of the prevalence of the imbalanced nature of network data can be seen when reviewing Table 3 from Ring's survey, where he denotes if a dataset is balanced or not across 34 surveyed datasets [84]. In that analysis, only one dataset [15] is indicated as being balanced, though that dataset is not publicly available. Similar results can be seen when reviewing the statistics of benchmark datasets.

The issues introduced due to imbalanced data for cybersecurity are similar to the issues found in other domains with imbalanced data. Namely, training on the imbalanced data tends to result in a model that performs extremely well on the majority class, while performing poorly on the minority class [42]. Unfortunately, this is an intrinsic property of cybersecurity data that researchers must deal with in order to optimize classifier performance. In general, one can leave the data as-is, use an undersampling technique, or use an oversampling technique.

Network Drift

We use the term network drift to refer to concept drift as it affects data in the area of cybersecurity. Concept drift occurs when the context of our data changes in relation to our predicted target [105]. In practical terms, this means the data we used to train a machine learning model is no longer applicable to the current state of our novel samples of data in relation to making a prediction. For this discussion, we break down the challenges posed by concept drift when applying machine learning to cyber defense data into three categories: In-Network Drift,

Drift Across Networks, and Evolving Attacks.

For In-Network Drift we consider a single network and its behavior over time. It has been shown that in the general operating of a given network, there are temporal drifts in normal network activity [65]. This occurs on various time scales such as daily, weekly, and seasonally [65].

This property of network data complicates applying machine learning techniques to cybersecurity as the training data must account for these patterns in order to identify benign behavior from attack behavior. As an example, consider a routine audit of a network which takes place at the same time each week, in which machines are tested for security policy compliance for which ports should be open. At that time of the week, we would not want a machine learning model to flag an alert for a port scan attack as it is a routine audit. However, if the same behavior occurs at a different time, we would want the model to flag it as suspicious activity.

When referring to drift across networks, we consider the fact that each network is unique and will have different network traffic patterns that would be considered normal behavior. This provides a burden on the application of machine learning to cybersecurity as one can train a model on a single network or public benchmark dataset, but truly have little idea of how generally applicable that model is across different networks. This issue is magnified for any machine learning model that unwisely utilizes IP addresses as features, as these will vary drastically from network to network in both value and behavior. Recent research seeks to improve this situation through the use of techniques aimed to standardize some differences across networks such as using Feature as a Counter (FaaC) [66] and other transfer learning techniques [109].

The final area of drift for discussion comes in the area of the attacks being

launched against systems. One of the main reasons researchers have looked to apply machine learning to cybersecurity is that signature-based methods are beginning to fail to be effective against evolving attacks, and provide little help against zero-day attacks [108]. For supervised machine learning methods employed for cyber defense, this requires periodic retraining and the availability of data which contains up-to-date attack samples. This issue is less severe for machine learning techniques that are semi-supervised or unsupervised, where anomaly detection methods will have a leg up on detecting zero-day attacks compared to supervised methods [98].

Heterogeneous Data

Earlier in this chapter we indicated that the native collection formats for cybersecurity data, and network data in particular, is a binary format. This is not unlike other areas where machine learning is applied, however, the format requires specialized tools such as Wireshark[6] or Zeek[7] to read the data. Additionally, within the binary format, is heterogeneous data of both statistical measures and categorical data that must be accounted for when performing machine learning. This property of the data provides another area of challenge when applying machine learning to cybersecurity, and we discuss the specifics here.

There are numerical features associated with cybersecurity data that include items like MTU size, packet size, bytes transferred, and number of packets in a given network flow. These are relatively easy to deal with in a fashion similar to other areas where machine learning is applied. When dealing with NetFlow data, however, one can observe that the range of certain values such as bytes transferred

[6]https://www.wireshark.org/
[7]https://zeek.org/

can vary wildly depending on network activity. While generally a standard practice, some sort of normalization or scaling to the values is usually employed.

The more complicated aspect of dealing with cybersecurity data occurs when we introduce the categorical data that is included. This includes items like protocol, TCP flags, IP address, and port number. While these are represented as numbers, the numbers provide little meaning to what they indicate functionally or in relation to each other. One can consider port numbers as an example of the complications they pose for machine learning. Port numbers generally range from 0 through 65535 on most systems. This makes it untenable to simply one-hot encode the values seen in cybersecurity data. Additionally, the values of two port numbers do not indicate how closely related their functionality may be. So if these are to be used for machine learning, one must find a proper way to encode this information while remaining faithful to the relationships of the ports. Similar complications arise when considering IP addresses and the relationships they represent on a given network.

We can contrast this to another field which uses binary formats such as image processing. In this field, all the data of an image represents a value corresponding to a certain color. Using image data as input to a machine learning algorithm, one can simply take the image "as-is" or perform some simple normalization on the image to get a meaningful representation.

Variability of Network Data

As indicated by Sommer, the input data used when applying machine learning to cybersecurity contains "enormous variability" [97]. A given network, within its normal operations, can exhibit unpredictable behavior over short periods of time. This includes periods of unexpected bursty traffic or even unexpected lulls

in traffic. The variability extends to the type of network traffic taking place as well; protocols being used will vary over time in an unpredictable manner [97].

One of the main drivers for this is the fact that the data seen across a network is driven by a combination of software and human initiated activities. As the goals and interests of the users of a network change, so will the nature of the network traffic. This becomes more widespread the larger the network is in terms of both infrastructure and users.

The implications of this variability mean that machine learning models need enough data that represents normal network activity to attempt to account for this variability. Ideally, we would want a machine learning model to ignore some "benign anomaly" on the network but flag one that is indicative of an issue or attack. This likely means multiple extended periods of data collection on a *specific* network for which machine learning is going to be applied. Alternatively, some recent works seek to combine data from multiple networks to attempt to achieve more robustness against this issue [109, 91].

2.1.4 Host Intrusion Detection

Host intrusion detection (HID) has similar detection goals as NID, however, it is concerned with activity within a single host. In this environment, when performing HID, we have access to the local host's network activity, system logs, software files, and software behavior that can be analyzed. An example of parts of a host intrusion detection system (HIDS) would include a local firewall or a user's antivirus software. As with NID, both signature-based and anomaly-based methods are commonly employed for HID.

2.1.5 Malware Detection and Analysis

While it could be considered a part of HID, we provide a distinct description for malware detection and analysis. Distinctly, malware detection and analysis is often done either on a live system or in an environment specifically set up for analyzing malware, such as a virtual machine or honeypot. Classically, antivirus software would attempt to detect malware in a signature-based manner, however, we are now seeing methods employed based on software behavior, system logging, or deep learning methods which analyze binary files [28].

2.2 Anomaly Detection

To begin discussing anomaly detection one must first define what is meant by an anomaly and more specifically, what constitutes an anomaly. A general definition is that anomalies are patterns found in data that differ from some defined concept of patterns that are considered normal [22]. We draw a distinction from concepts such as noise and outliers found in data in that an anomaly provides some interesting insight into the data that we would want to discover [22]. Additionally, we distinguish anomalies from novelties found in data in that novelties are the initial emergence of new normal patterns, though we may not know this at the time of their discovery [22].

There are generally three types of anomalies to consider: point anomalies, contextual anomalies, and collective anomalies [4]. With point anomalies, a single sample of the data deviates from the normal pattern one would expect [4]. As an example, the daily average high temperature for the end of December in Massachusetts is 39°. One would consider a day in the end of December with a temperature of 70° to be a point anomaly. Contextual anomalies are data samples that are

only considered anomalous due to a particular context or dependency [4]. One example of a contextual anomaly would be a severely overcrowded supermarket that is not close to a popular holiday. This same type of crowd would not be considered an anomaly close to a popular holiday as the crowd would be expected close to the holiday. A collective anomaly occurs when a particular pattern (or similar patterns) in the data are considered anomalous when grouped together [4]. As an example, it is not completely uncommon for a single stock to crash on a given day (though it would likely be a point anomaly), however, if a complete market crash occurred and all stocks crashed on a given day we could consider that a collective anomaly.

The method used for detecting anomalies is generally determined by the availability of labels within a given problem domain [20]. If completely labeled data is available, for both normal samples and anomalies, one could effectively apply supervised anomaly detection methods [20]. This is rare for most problems in anomaly detection and so more emphasis in the research is placed on semi-supervised learning and unsupervised learning. Semi-supervised learning can be applied when we have labeled (or known) normal samples of data. With many of these types of techniques, one can obtain a representation of "normal" for the data and then use that model with new samples to detect anomalies by how much they deviate from this model. When no labels are available, unsupervised anomaly detection is employed. The general assumption with these methods is that the data will be dominated by normal samples [4]. When this assumption is untrue, a lot of false positives will likely occur with these methods [4].

There are two main types of output for an anomaly detector: anomaly score and label [4]. The anomaly score is some value associated with how likely a given data

sample is an anomaly. Often, a threshold is applied to this score, where if the score is greater than the threshold then the anomaly detector flags the sample as anomalous. When an anomaly detector outputs a label it is generally a binary output that indicates if a sample is an anomaly or normal. When applying anomaly detection to NID, we make the assumption that attack network data will be anomalous compared to benign network data.

2.3 Machine Learning Algorithms

In our work we sought to pair our techniques with many different downstream classifiers in order to gain confidence in its general effectiveness. In this section we provide background for the machine learning algorithms utilized throughout our research. We go into detail for the algorithms that are significant for their application to NID or potential for practical application beyond research. For the remainder we provide a lighter treatment as they would be less likely candidates for practical applications beyond our own empirical research studies.

2.3.1 Random Forest

Random forest is a classic supervised learning algorithm that uses an ensemble of decision trees to arrive at a prediction. While they can be used for both regression and classification, they are generally used for classification when applied to NID. When constructing a single decision tree for classification, we perform recursive binary splitting such that a prediction from a given region will align with the majority classification label within that region. Each split is made according to its purity, the number of samples within a split that belong to the same class. This is measured either by the Gini index or entropy as shown in equations 1 and 2

respectively [46].

$$G = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{1}$$

$$D = \sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk} \tag{2}$$

In these equations we assume that there are K classes and that $\hat{p}_{mk}$ is the proportion of training samples of the kth class in the mth region of the tree [46]. The split is made where the values of the measures are smallest, as that is an indicator of high purity [46].

Random forest improves upon a traditional decision tree by using predictions from an ensemble of decision trees. When constructing the trees for the forest, each split of a given tree only considers a random sample of features to use to make the split. This increases the diversity of the trees in the random forest often leading to stronger predictions.

When applied to NID, random forest is overall a well-performing algorithm [75, 9]. This has held true within our own work, where its performance on recent datasets was generally best out of all traditional machine learning algorithms. The main reason for its inclusion as a key machine learning technique for NID is that despite being a traditional algorithm, it often performs better than neural networks [9]. In addition, we can normally get a respectable baseline of the complexity of a given NID dataset by evaluating it with a random forest model. For these reasons researchers continue to use random forest either on its own, with some variation of tree generation [47], or as part of a larger pipeline of models [9].

2.3.2 Naive Bayes Classifier

The naive Bayes classifier is a supervised method based on probability theory and Bayes' theorem [46]. This classifier minimizes classification error by assigning each test sample to the class which it most likely belongs using its features in comparison to training data [46]. Using the conditional probability in equation 3 we assign a given sample to the class j for which the probability of the sample belonging to that class is greater than 0.5 [46]

$$\Pr(Y = j|X = x_0) \tag{3}$$

where Y is either zero indicating a benign sample, or one, indicating an attack sample, and x_0 represents the features of our test sample. The Bayes classifier was used for benchmarking purposes with an assumption that the likelihood of features for a given sample follows a Gaussian distribution in our work with classical algorithms, however, it is not prevalent in current research within the field of network intrusion detection.

2.3.3 Logistic Regression

Logistic regression is a supervised classification algorithm that adapts the concepts of linear regression such that they can be applied to the classification setting [46]. Taking the typical linear regression model, $p(X) = \beta_0 + \beta_1 X$, the main issue that can occur is that our probabilities may be negative or greater than one. For this reason, logistic regression applies the logistic function, a sigmoid function, as shown in equation 4 [46]

$$p(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}} \tag{4}$$

which constrains the probabilities to be between zero and one. We fit the parameters of the function using maximum likelihood and then use the learned parameters during inference. In our work we used logistic regression as one of our comparison classical machine learning algorithms, however, it is often insufficient for use in network intrusion detection and is not seen often in this area of research.

2.3.4 K-Nearest Neighbors

K-Nearest Neighbors (KNN) is a supervised classification algorithm that attempts to estimate the Bayes decision boundary [46]. It does this by taking a novel sample, and then considering the K closest neighboring samples, which we identify as $\mathcal{N}_0$. As shown in equation 5, we classify the sample using the label of $\mathcal{N}_0$ with the highest probability [46].

$$\Pr(Y = j|X = x_0) = \frac{1}{K} \sum_{i \in \mathcal{N}_0} I(y_i = j) \tag{5}$$

In our work we used K set to values of five and ten when exploring classical algorithms. While KNN often produced results comparable to random forest in our results, it is not considered a primary classifier when working with network intrusion detection due to its reliance on labels and sensitivity to training data.

2.3.5 Support Vector Machine

The support vector machine is a supervised machine learning algorithm constructed from a generalization of the support vector classifier. When used for binary classifi-

cation, the main idea behind a support vector classifier is to construct a hyperplane such that if a sample falls on one side of the hyperplane, it is assigned to one class. Falling on the other side of the hyperplane, the other class is assigned to the sample. We call the hyperplane a maximal margin hyperplane as we want there to be the maximal amount of distance between the hyperplane and the closest samples of data. These closest samples of data that define the hyperplane are referred to as support vectors. One can define the solution to a support vector classifier using equations 6 through 9 [46].

$$\underset{\beta_0,\beta_1,...,\beta_p,\epsilon_1,...\epsilon_n,M}{\text{maximize}} \; M \tag{6}$$

$$\text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1, \tag{7}$$

$$y_i(\beta_0 + \beta_1 x_{i1} + \beta_2 x_{i2} + ... + \beta_p x_{ip}) > M(1 - \epsilon_i), \tag{8}$$

$$\epsilon_i \geqslant 0, \sum_{i=1}^{n} \epsilon_i \leqslant C, \tag{9}$$

In these equations, we have n samples denoted as x with p dimensions and β representing the hyperplane's parameters. M is the margin we are maximizing with ϵ allowing for slack such that some observations can be on the wrong side of the hyperplane constrained by a non-negative tuning parameter C.

The support vector machine generalizes these equations to allow for the use of kernel functions which map to a higher dimensional space and can create non-linear hyperplanes represented by equation 10 [46]

$$f(x) = \beta_0 + \sum_{i \in S} \alpha_i K(x, x_i) \tag{10}$$

where S is the set of support vectors and K represents a kernel function that maps to a higher dimensional space. Popular kernel functions include a polynomial kernel or radial kernel [46].

2.3.6 One-class Support Vector Machine

One-class support vector machine (OCSVM) is an algorithm that adjusts the previously discussed traditional support vector machine to operate in an unsupervised manner [92]. The main concept for a OCSVM is to find a hyperplane such that it has a maximum margin that separates normal samples from anomalies [92]. The algorithm does this by minimizing equation 11 [92]

$$\underset{w,\xi,\rho}{min} \frac{1}{2}||w||^2 + \frac{1}{vn}\sum_i \xi_i - \rho \tag{11}$$

$$\text{subject to } (w \cdot \phi(x_i)) \geqslant \rho - \xi_i,\ \xi_i \geqslant 0 \tag{12}$$

where n is the number of samples, ξ are slack variables, ρ is the distance to the origin, and w is a vector perpendicular to the hyperplane in a high-dimensional feature space mapped by some function ϕ, and v is a parameter to control the trade off between the two terms of the equation. Once fit, the OCSVM determines outliers using $f(x) = sgn((w \cdot \phi(x)) - \rho)$ such that inliers are positive with outliers being negative. Similar to the traditional support vector machine, the OCSVM supports the use of various kernel functions for introducing non-linearity to the hyperplane. In our work, we seek to fit the OCSVM using only benign network

flow samples such that attacks will be considered anomalies.

2.3.7 Isolation Forest

Isolation forest is an unsupervised algorithm first introduced as a way to perform anomaly detection without depending on measures of density or distance among training samples [61]. Consisting of an ensemble of binary trees referred to as isolation trees, the main insight of the technique is that anomalies should be more prone to isolation; separation from other samples, compared to those which exhibit normal behavior [61]. In terms of a single isolation tree, a normal sample would require more partitions in the feature space to be isolated compared to an anomaly. This quality allows one to construct an anomaly score using the average height of the trees in an isolation forest as shown in equation 13 [61]

$$s(x, \psi) = 2\frac{-E(h(x))}{c(\psi)} \tag{13}$$

where x is the input sample, ψ is number of samples drawn from training data used to construct each isolation tree, $E(h(x))$ is the mean path length of sample x across the isolation trees in the forest, and $c(\psi)$ is a normalizing factor calculated based on the given sampling size ψ [61]. In our work, we fit isolation forests using only benign network samples. In doing this we expect that attack network flow samples will be easier to isolate compared to benign network flow samples, leading to a higher anomaly score for those samples.

2.3.8 Local Outlier Factor

The local outlier factor (LOF) algorithm has two differences compared to most other anomaly detection methods. Namely, it provides a score for how much a

given sample is outlying, called the local outlier factor, and determines this score based on how isolated the sample is compared to its local neighborhood of samples [17]. One advantage of this algorithm is that by dealing with the locality of points, it avoids issues related to only using global densities of the data, which can cause outliers to be misclassified under certain conditions [17]. The local outlier factor of a given sample p would be determined using equation 14 [17]

$$LOF(p) = \frac{\sum_{o \in N} \frac{lrd(o)}{lrd(p)}}{|N(p)|} \tag{14}$$

where N is the sample's local neighborhood and lrd is the local reachability density function outlined in the original work [17]. This calculation provides the ratio between a given point's local reachability density compared to that of its local neighborhood [17]. When sample p is an inlier, this value will be close to one, while outliers would have a value greater than one. In our use of the algorithm, we expect that attack network flows would have a larger local outlier factor than benign network flows.

2.4 Feed Forward Neural Networks

We refer to the trainable models used in deep learning as neural networks. While these networks come in many shapes and sizes they all typically form some parameterized function that seeks to approximate an unknown true function based on provided training data. In general, the structure of neural networks can be described using six parts: input layer, hidden layer(s), output layer, activation functions, cost function, and an optimization function. Figure 4 provides a high-level view of a feed forward neural network and its structure. Deep learning models

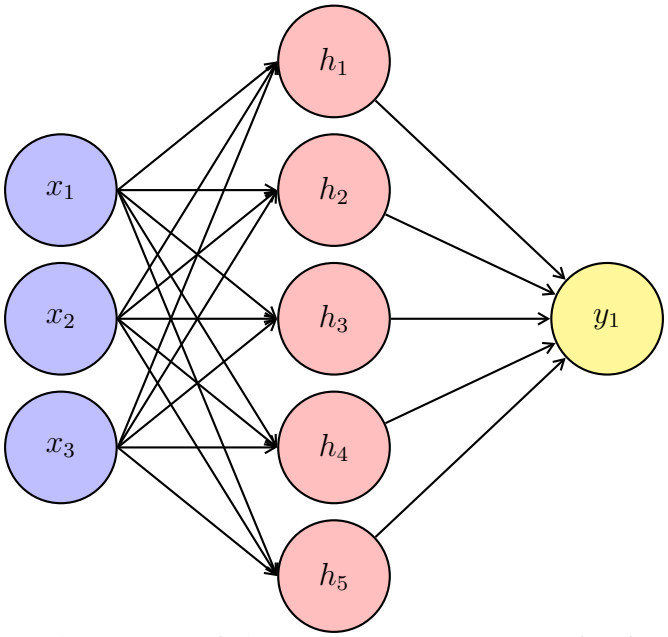


Figure 4: Sample architecture of the main components of a feed forward neural network. Depicted in blue are the input features. Depicted in pink are nodes of a hidden layer. This network has a single output y_1 denoted in yellow.

are normally trained using some form of gradient descent derived optimization algorithm [39]. As input flows through a neural network, we call this forward propagation, which has a loss calculated by a model's cost function. The gradient of this loss is then calculated back through the network in a stage called back propagation. This gradient is then used to update the trainable parameters of the model.

Non-linearity for a deep learning model is introduced by non-linear activation functions that are used between layers of a network. Common activation functions used include the Rectified Linear Unit (ReLU), sigmoid, and the hyperbolic tangent (tanh) function.

In feed forward networks a sample is provided to the model at the input layer with a node for each feature of the sample. The data is then passed through the network between connections of each node of a previous layer to the next layer.

In general, an activation function is present between each layer. The number of layers, nodes, and activation functions to use are generally determined empirically based on validation data performance of the model. The layers of a feed forward neural network typically perform an affine transformation defined as $h = g(Wx + c)$ where x is the input, W are associated weights, c is a vector if biases, and g is our selected activation function [39]. Given this we can represent a full neural network with m layers using equation 15

$$f(\theta, x) = g(W_m...(g(W_2 g(W_1 x + c_1) + c_2)... + c_m) \tag{15}$$

where θ are all of the weights and biases learned by the network based on training data. While different loss functions can be used such as mean-squared error (MSE) or binary cross entropy, the loss calculated after forward propagation can generally be represented using equation 16

$$Loss = \ell(y - f(\theta, x)) \tag{16}$$

where y represents some target for the output of the network and ℓ is the loss function. During training, the gradient of this loss is then used to update our parameters θ as shown in equation 17

$$\theta_{new} = \theta - \alpha \nabla Loss \tag{17}$$

where $\nabla Loss$ is the gradient of our loss and α is a hyperparameter referred to as a learning rate, which controls the magnitude of the movement of the parameters during each step of training. While the specific criteria can vary, we generally stop training once convergence of the parameters occurs resulting in little change from

one round of training to the next.

In our work, we utilize feed forward neural networks for classification. Additionally, as discussed in Section 2.6, the core structure of our method can be considered a specialized format of a feed forward neural network.

2.5 Recurrent Neural Networks

One of the limitations of feed forward neural networks is that each sample is processed through the network individually such that information from previous samples are lost. Recurrent neural networks (RNNs) were created as a way to provide some memory to a neural network such that they can better perform inference on datasets with sequential dependencies [85]. Depicted in Figure 5 in both a rolled and unrolled fashion, one can see that these networks consider samples at some time t. The input to the RNN at a time t consists of the state of the network from time $t-1$ and the current input X_t [85]. This is visualized in Figure 5 with the specific calculation used in this work provided in equation 18 [85]

$$h_t = \tanh(X_t U^T + h_{t-1} V^T + b) \tag{18}$$

where X_t is the input at timestep t, U are the weights associated with the input, h_{t-1} is the hidden state of the previous time step with V being the respective weights for the previous hidden state, and b represents the biases. The output can then be calculated using equation 19 [39]

$$y_t = h_t W^T + d \tag{19}$$

where h_t is defined by equation 18 with W and d being the weights and bias asso-

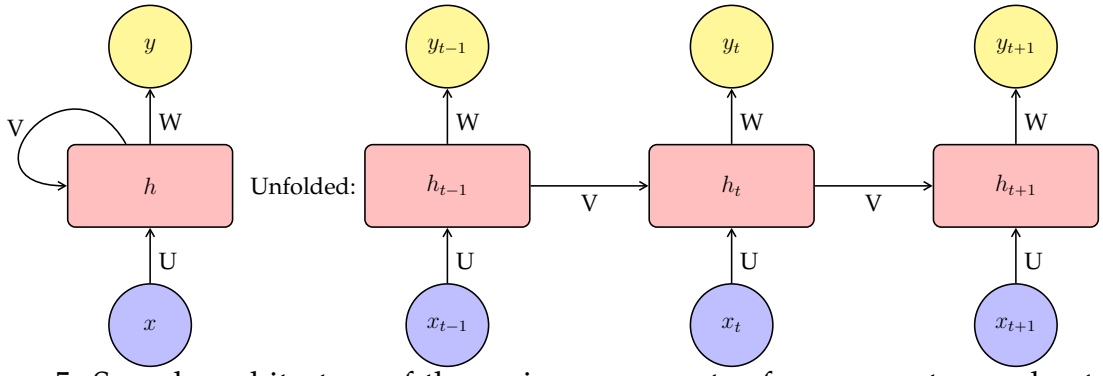


Figure 5: Sample architecture of the main components of a recurrent neural network. Depicted in blue are the input features. Depicted in pink are recurrent hidden nodes. The network's output from each hidden node is denoted as y in yellow.

ciated with the output.

Similar to feed forward networks, RNNs utilize a loss function and a gradient descent based backpropagation optimization, backpropagation through time (BPTT), to perform learning of the network parameters. There are several variants to RNNs such that they can provide an output at each time step or a single output after a sequence has been processed as used in this work.

2.5.1 Long Short Term Memory

In additional to a classic RNN, Long short term memory (LSTM) networks were created to overcome issues encountered when training RNNs such as vanishing and exploding gradients [44]. In addition, they have shown the ability to successfully perform inference on tasks that require long-lasting memory [39]. The main differences between the RNN depicted in Figure 5 and LSTMs, is that LSTMs include cells with self-loops and a series of gates that control the flow of memory through the network [44]. These gates, which are controlled by learned parameters of the network include the input gate, forget gate, and output gate. For each

cell, the input gate controls if a given feature will be accumulated within the cell. The state unit of a cell contains the self-loop which is controlled by the forget gate. The output gate determines how much of a given cell's state, c_t, will be output. Each of these gates are generally neurons with learned parameters and a sigmoid activation function. The relevant LSTM calculations are shown in equations 20 through 25 [87].

$$i_t = sigmoid(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \tag{20}$$

$$f_t = sigmoid(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \tag{21}$$

$$g_t = tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \tag{22}$$

$$o_t = sigmoid(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \tag{23}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t \tag{24}$$

$$h_t = o_t \odot tanh(c_t) \tag{25}$$

In these equations i_t, f_t, o_t are the input, forget, and output gates respectively; g_t is the cell activation; c_t is the cell state at time t; h_t is the hidden state at time t; h_{t-1} is the hidden state of the previous time step; and c_{t-1} is the cell state at the previous time step. Additionally, $\odot$ is the Hadamard product with W and b representing the various weight and bias vectors of the network. It should be noted that in our work we do not use the LSTM calculations that make use of projections as outlined in the proposed LSTM architecture of [87]. While this makes the LSTM simpler in structure, it does not impact our main goal of comparing its performance between different sets of input features.

2.6 Autoencoders

Autoencoders are a commonly used form of feed forward neural network with properties that lend themselves to a number of applications such as data generation, feature selection and reduction, denoising, and anomaly detection [39]. As autoencoders are a specialized form of neural network, the general components and properties described in Section 2.4 apply to these networks as well. With the goal of reconstructing the original input, the useful properties of autoencoders are a result of its unique structure consisting of an encoder with a bottleneck layer and a subsequent decoder [54]. The encoder consists of one or more hidden layers and takes as input the original input features. The final layer in the encoder, commonly referred to as the bottleneck layer, contains fewer nodes than the original input [54]. The decoder then takes as its input the output of the encoder and reconstructs the original input. The key part of the autoencoder structure is the bottleneck layer, as it forces the model to learn a compressed representation of the original data and presumably learns the most important characteristics of the original input [54]. The structure of a basic autoencoder can be seen in Figure 6.

In our work we refer to the original input as X and define L as the autoencoder reconstruction using equation 26

$$L = D(E(X)) \tag{26}$$

where E is the encoder, and D is the decoder. While training an autoencoder, the goal is to minimize the difference between the original input and the autoencoder reconstruction. Rewriting equation 16 for an autoencoder yields equation 27 which can be simplified to our notation shown in 28.

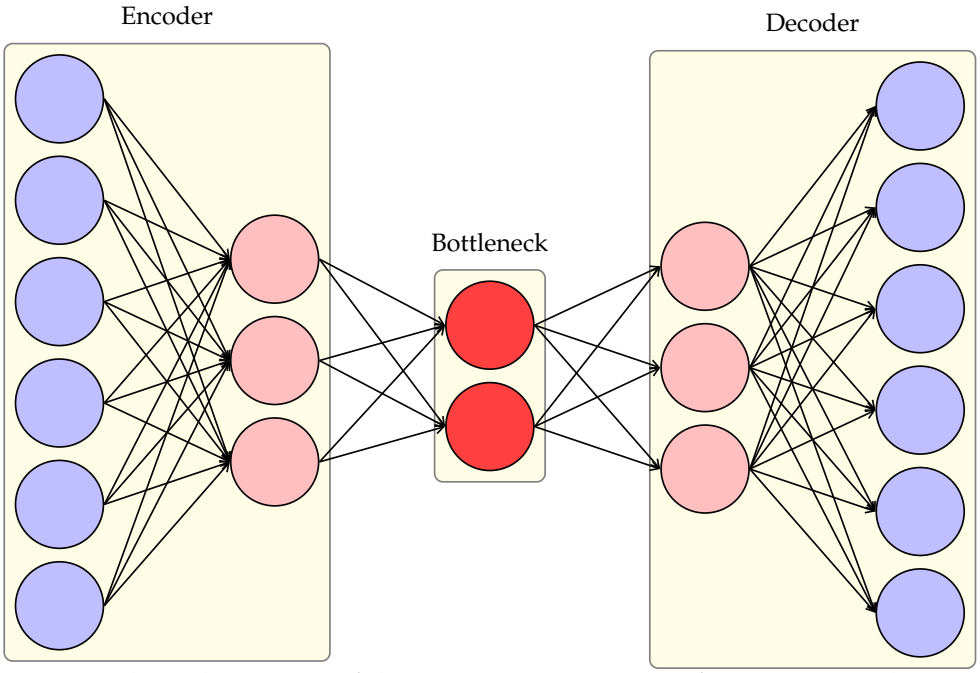


Figure 6: Sample architecture of the main components of an autoencoder. Depicted in blue are the input features and their respective reconstructions. Depicted in pink are hidden nodes. Depicted in red is the bottleneck layer.

$$Loss = \ell(X - D(E(X)) \tag{27}$$

$$Loss = \ell(X - L) \tag{28}$$

Common loss functions used when training autoencoders include MSE or Kullback-Leibler (KL) divergence. In our work we typically use MSE for our loss function.

3 Related Work

3.1 Learning Techniques Applied to Network Intrusion Detection

There is a wealth of learning techniques employed for NID that include statistical methods, classic machine learning, and deep learning. In this section we provide a review of the literature with a concentration on learning methods, with a focus on deep learning methods, which have begun to dominate the research space in this domain.

One recent work uses statistical analysis of network data to determine a normal network profile using only benign network data [71]. In that work, the authors determine the probability density function (PDF) of a Drichlet Mixture Model using the benign data during training. During testing, the lower-upper interquartile range is used as a threshold on new samples to detect attacks. A similar work uses statistical methods for feature selection where multiple correlation matrices of network features are used to generate features and a profile of normal network behavior [40]. Anomalies are detected by comparing the statistical features of new samples to the normal profile using the Mahalanobis distance.

Traditional machine learning models remain a popular technique in current literature. Random forest is often used for initial benchmarking of datasets as it has been shown to generally perform well for network intrusion detection in particular [75, 9]. Despite being a traditional machine learning algorithm, researchers have shown it to outperform neural networks for some NID datasets [9]. In recent research we see variations on how the trees in a random forest are generated [47] or it is used as part of a larger pipeline of models [9].

The main limitation to many classic machine learning algorithms is the fact that many of them require supervised learning which produces models that struggle with zero-day attacks. For this reason, researchers tend to favor algorithms that can be trained in an unsupervised manner or using only benign network data. As an example, OCSVM was shown to perform well on network intrusion data and mitigates the issues surrounding the scarcity of real-world attack data [37]. Several recent works incorporate OCSVM models into a larger pipeline of models with success detecting network attacks [67, 3]. Other works concentrate on augmenting OCSVM to optimize its parameters in the context of network intrusion detection [86, 36].

More prominent in recent literature are learning techniques that incorporate neural networks and deep learning. Feed forward neural networks are often employed to perform classification on network data [64]. For example, one recent work focuses on taking advantage of feed forward neural networks specifically for software defined networks [100]. In addition to feed forward networks, recent literature also applies convolutional neural networks (CNN) to network intrusion detection in order to find localized and correlated features. One such work compares a CNN's performance to a traditional multilayer perceptron (MLP) and shows that it outperforms the MLP for NID [5]. Another work presents the HCRNNIDS model, which uses a CNN in conjunction with an LSTM to successfully detect network attacks [50].

Perhaps one of the more striking shifts in the literature over the last decade is the emergence of taking advantage of the sequential nature of network data by using models such as RNNs, LSTMs, and Gated Recurrent Units (GRU). The DDoSNet model combines the architectures of an RNN with an autoencoder and utilizes

the bottleneck layer to classify distributed denial of service (DDoS) attacks using network flow data [30]. The recent work by Ullah and Mahmoud performed extensive analysis of RNNs, GRUs, and LSTMs to show their suitability for network intrusion detection specifically for Internet of Things (IoT) networks [103].

3.2 Feature Generation Techniques for Network Intrusion Detection

In this section we review recent works related to feature generation when applying machine learning to network intrusion detection. We reserve feature generation techniques that utilize autoencoders to Section 3.3 as it is one of the common usages of autoencoders in this space. Finding optimal features for learning models applied to network intrusion detection is a perennially present focus in the literature. One recent work combines an interesting ensemble of models and techniques to perform feature generation using NetFlow data [81]. An ensemble of recurrent models consisting of a RNN, GRU, and LSTM are trained, and dimension reduction is performed on their output features using kernel principal component analysis (KPCA). The resulting features are then combined and provided to several downstream classifiers such as random forest.

The work that presents the full NIDS called Kitsune also contains a unique feature generation component [69]. Kitsune calls this component its feature mapper, and it utilizes agglomerate clustering to divide the original feature space into non-overlapping subspaces. Each subspace is then provided to its own set of downstream models to perform attack detection. A similar work takes an original feature set from NID datasets and uses a greedy recursive feature addition algorithm to fine-tune their final features used in downstream classifiers [62].

Another approach for feature generation comes from Sarhan et al. where they promote using a standard set of features to be used when applying learning algorithms to NID [91]. In this work, features are generated based on the NetFlow v9 standard such that the data should be available in the vast majority of network settings in a well-defined manner.

3.3 Autoencoders Applied to Network Intrusion Detection

In this section we cover recent applications of autoencoders for NID. We can categorize the general usage of autoencoders for NID into several broad categories based on two of the key components of the autoencoder model as shown in Figure 7. In one case when applied to NID, we often see the latent layer of autoencoders used either for feature reduction or feature selection. In the other general case, we see the reconstruction loss from an autoencoder used to detect network attacks or as part of a feature generation process. In the former, the reconstruction loss is generally transformed using an aggregation operation such as calculating the MSE of a NetFlow sample. On the whole, we make the observation that in most cases, autoencoders are used for the general purpose of anomaly detection when applied to NID.

3.3.1 Using the Autoencoder Latent Layer

The general idea of using the autoencoder bottleneck layer for either feature reduction or feature selection stems from the insight that it captures a compact encoding of the salient parts of the original features. For this reason, many recent works concentrate on this technique for NID [18, 110, 74, 104]. One recent work is dedicated to analyzing the effects of the bottleneck size when using autoencoders for NID

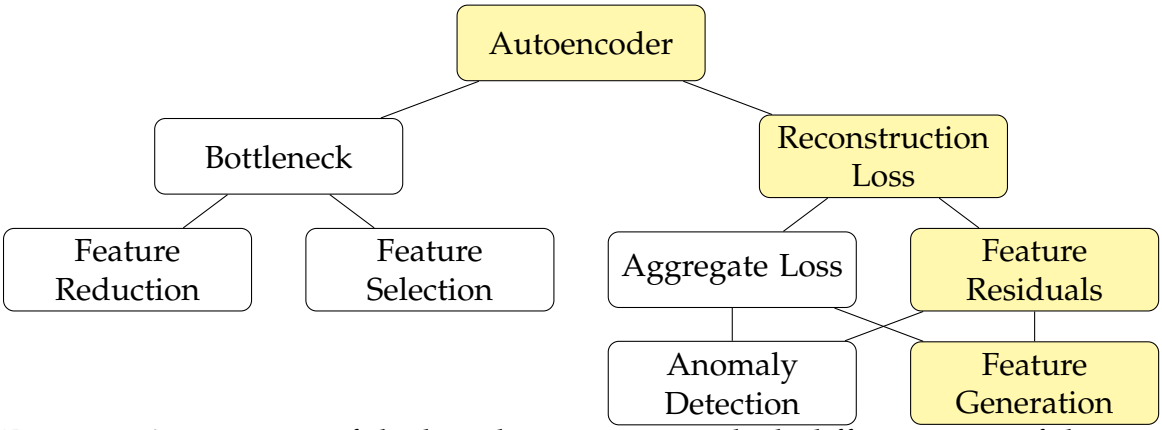


Figure 7: A taxonomy of the broad categories in which different parts of the autoencoder are used. Traditionally the bottleneck, or latent layer, of the autoencoder is used for various feature reduction or feature selection tasks. The reconstruction loss of an autoencoder is generally used as an aggregate statistic to perform anomaly detection with a threshold. Some works use the reconstruction loss as a way to generate features for downstream tasks, again, generally using an aggregate statistic to characterize the loss. Highlighted in yellow is the category in which our technique falls. While we display a category for using feature residuals directly for anomaly detection it is rarely seen in practice.

[98]. Here, the authors varied the bottleneck size and used a z-score threshold of the reconstruction error, an aggregate residual metric, to evaluate its effects. This work found that often times very small bottleneck sizes can be used depending on the NID dataset being evaluated.

An interesting modification to this technique in recent literature adds a dropout layer after the bottleneck layer during training [33]. It was shown that doing this during training improves the robustness of the encodings in the latent layer which can then be used as features to downstream models effectively. Other works that modify the autoencoder structure to enhance the efficacy of the bottleneck layer include the DDoSNet model [30]. In that work the autoencoder layers are modified from traditional affine layers to be recurrent layers in order to capture the sequential nature of network data. The bottleneck layer was then used as input to a final softmax layer for classifying network attacks.

The usage of the bottleneck layer is quite prevalent when applying NID to IoT networks. In these networks, there is generally less computing power available compared to traditional networks. By using the bottleneck features of an autoencoder for downstream processing, the compute power needed for inference can often be reduced [56, 63, 14]. One unique application of this technique for IoT modifies the traditional training process for an autoencoder by constructing two bottleneck layers. One bottleneck is trained using only benign samples, while the other is trained using both attack and benign samples. A filtering process is then used to determine which features are critical to discriminating attacks from benign network samples. This reduced feature set is then used for inference in combination with a downstream classification method.

3.3.2 Using the Autoencoder Reconstruction Loss

When using autoencoder reconstruction loss to identify network attacks the general idea is to train the autoencoder on only benign network data. In doing so, the assumption is that the autoencoder will struggle to reconstruct attacks and have a higher resulting reconstruction loss on those samples. Within this general framework, many works in this area are unique in their autoencoder architectures, how they determine their threshold, or the actual aggregate loss metric used for detecting anomalies [7, 11, 23, 111, 38]. One recent work that uses this general framework focuses on cyber-physical environments and combines lightweight features while training the autoencoder using only benign samples [76]. A threshold is then applied to the MSE of the resulting autoencoder reconstructions to detect anomalies. A similar work uses this same concept as part of a larger two-stage system to detect network attacks [112]. In the first stage of the system, the authors use a light

gradient-boosting model to perform initial detection of network attacks. An autoencoder is then used in the second stage to confirm any network flows identified as benign also have a reconstruction MSE that is below a threshold to confirm the original decision.

Another recent trend in the literature uses an ensemble of autoencoder reconstruction loss to detect network attacks. Discussed previously, Kitsune trains an autoencoder for multiple sub-spaces of the original set of features. A final anomaly detection process is performed by using the MSE from each autoencoder along with a threshold operation. Another work uses a similar ensemble of autoencoder reconstruction loss as the third step in a three-step anomaly detection system [62].

Several additional works focused on IoT networks train an autoencoder for each device in order to obtain a benign profile for each device on the network [68, 29]. This helps alleviate issues with creating such a profile on networks with a high amount of heterogeneous devices. The reconstruction loss from each device is then used to evaluate if network samples are attacks. The proposed Auto-IF NIDS uses the MSE of the autoencoder loss to divide IoT traffic into attack and benign samples. Two isolation forests are then used as a second measure to further confirm that the benign and attack traffic has been correctly identified.

We now explore recent works that augment an original feature set by utilizing autoencoder reconstructions and their loss for NID. One recent work takes the autoencoder reconstruction and generates their final feature set by performing a sum operation with the original features [107]. AIDA is a full NIDS introduced to detect network attacks which uses a single autoencoder reconstruction loss metric for each sample [8]. This single metric is then used as a single additional feature to augment their original feature set which is then provided to a downstream clas-

sifier. Similarly, another work augments their classifier input feature set using a stacked sparse autoencoder to produce three additional features [113]. First, a bottleneck layer of a single node is used as a feature. The authors then use two forms of the reconstruction error from the autoencoder, Euclidean distance and cosine similarity, as the other two additional features.

There is little literature exploring the use of autoencoder feature residuals for network intrusion detection that we will develop in detail in Chapters 5 and 6. Among them includes the work that develops the DeepAID framework which has a goal of adding interpretability to deep learning-based anomaly detection systems [41]. In DeepAID, the autoencoder feature residuals are not actually used for anomaly detection. That work takes already identified anomalies and then uses the autoencoder feature residuals to interpret why they were classified as such. In our work, by using autoencoder feature residuals to actually identify network attacks, we get some of this interpretability for free by inspecting which features have high residuals. In other words, the feature residuals of attack samples with high magnitude identify the particular features that deviate the most from benign data.

One final work worth noting that makes use of autoencoder feature residuals is a work where the authors train an autoencoder on each device connected to a network [109]. A global classifier is then used to detect attacks from the network data with its input being the normalized autoencoder feature residuals from each device. While the focus of that work is on their entire NIDS, we differentiate our work by explicitly exploring the use of autoencoder feature residuals and capturing their performance across numerous robust conditions in order to show that they are generally applicable.

3.4 Robust Deep Autoencoders

In this section we discuss robust deep autoencoders (RDA) which served as the initial impetus leading our work towards using autoencoder feature residuals. RDA is a method with the goal of introducing the non-linearity data representation capabilities of deep autoencoders with the anomaly detection benefits of robust principal component analysis (RPCA) [77, 114]. RPCA takes an input matrix X and splits it into a low rank matrix L_{rpca} and a sparse matrix S_{rpca} such that $X = L_{rpca} + S_{rpca}$ [77, 114]. In this context, L_{rpca} contains a low-dimensional representation of X and S_{rpca} contains the outliers of the original matrix X [114]. To train the RPCA model we seek to minimize equation 29 subject to the constraints in equation 30.

$$\min_{L,S} ||L_{rpca}||_* + \lambda||S_{rpca}||_1 \tag{29}$$

$$s.t.||X - L_{rpca} - S_{rpca}||_F^2 = 0 \tag{30}$$

RDA adjusts this optimization by accounting for the use of an autoencoder for non-linearity. When detecting instance anomalies, one can use the minimization operation for RDA shown in equation 31 subject to the constraints in equation 32

$$\min_{\theta,S} ||L_{rda} - D_\theta(E_\theta(L_{rda}))||_2 + \lambda||S_{rda}^T||_{2,1} \tag{31}$$

$$s.t. X - L_{rda} - S_{rda} = 0 \tag{32}$$

where L_{rda} is the autoencoder reconstruction, S_{rda} consists of noise from the data,

λ is a hyperparameter controlling the sparsity of S_{rda}, and $D_\theta E_\theta(.))$ is the autoencoder. Of interest for the training process is that it must be trained in an alternating fashion using alternating direction method of multipliers (ADMM) where the autoencoder is optimized with S_{rda} fixed, while S_{rda} is optimized with L_{rda} fixed in an alternating cycle [114]. The resulting L_{rda} after the training procedure is complete, is a noise-free reconstruction of X, while S_{rda} contains the anomalous outliers and noise.

This technique was shown to provide anomaly detection capabilities on a number of benchmark datasets [114]. However, when implementing this technique across a variety of network intrusion detection datasets, it proved difficult to maintain consistent detection of network attacks and have a training process that converged consistently. For this reason we sought to relax the training process and pursue autoencoder feature residuals as discussed in detail in Chapters 5 and 6 while maintaining the spirit of having input data split into both an easy to reconstruct L_{rda} with anomalous data being captured into a separate S_{rda}.

4 Autoencoder Dimension Estimation Applied to Network Intrusion Detection

4.1 Autoencoder Dimension Estimation Overview

The technique of autoencoder dimension estimation (AEDE) was developed by Nahadur and Paffenroth as a way to estimate the intrinsic dimensionality of datasets with potentially non-linear dependencies [13]. Dimension estimation looks at a dataset and attempts to determine the number of latent factors it expresses. This intrinsic dimensionality is often less than the actual number of features present in the dataset due to a number of factors such as redundant features, locally correlated features, and dependencies between features [13]. Classic techniques such as principal component analysis (PCA) use orthogonal linear projections to perform both dimension estimation and dimension reduction, however, they are unable to accurately perform dimension estimation when the data contains non-linear dependencies. To account for this shortcoming, AEDE uses the non-linearity of the autoencoder and performs several transforms to its bottleneck layer in order to find a set of values analogous to singular values found by PCA, which are referred to as singular value proxies (SVP).

The main challenge that AEDE accounts for is enforcing both sparsity and consistency in the latent layer of the autoencoder which is required to perform DE [13]. To do this, AEDE applies two transforms to the latent layer of the autoencoder which both normalize the values and apply a sparsity penalty. First the values of the latent layer are processed through a sigmoid function and then normalized using equation 33

$$y^i_{norm} = \frac{y_i}{\sqrt{\sum_{i=1}^{k} y_i^2}} \tag{33}$$

where y_i is each node of the hidden layer which contains k nodes total. Next an $||.||_1$ penalty is applied to the normalized values to encourage sparsity of the values. This is represented in the objective function J_{sparse} for AEDE shown in equation 34

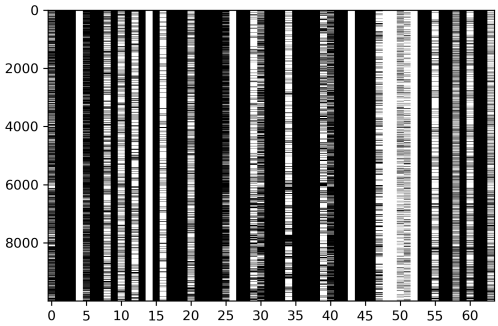
$$J_{sparse} = \ell(X - L) + \lambda \sum_{i=1}^{k} ||y^i_{norm}||_1 \tag{34}$$

where λ is a hyperparameter that controls the amount of sparsity applied to the hidden layer. Intuitively, as λ is increased we expect reconstructions to be less accurate as we are losing more data in the encoding.

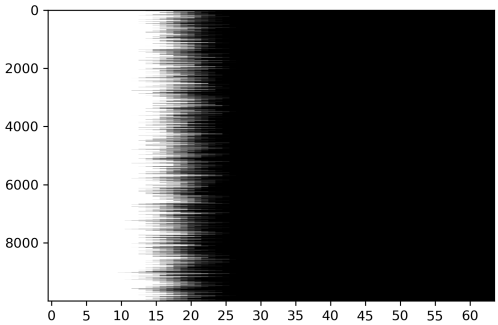
These transforms cause the hidden layer values to be both consistently in the range between zero and one as well as sparse. There are two remaining operations that are performed to finalize creation of the SVP dual to singular values. Shown in Figure 8, we first sort each row independently such that the highest values of each row are to the left. We then take the mean of each column to find our SVPs. To use the SVPs to determine intrinsic dimension we perform a threshold operation on them using a tunable parameter α.

4.2 Applications to Network Intrusion Detection

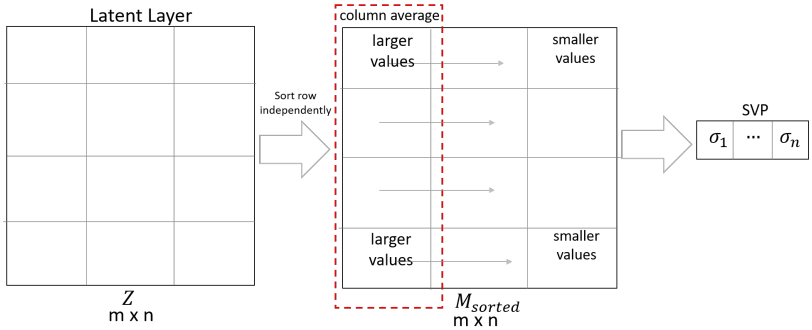
When applying this technique to NID, we asked the question: "Does network dimensionality change when the network is under a cybersecurity attack?". To answer this we utilized the CICIDS2017 NID dataset NetFlow data as described in Appendix A.4, using Monday's all-benign data for training and validation of the autoencoder while testing on the remaining scenarios. We used 70% of the benign



(a) Latent Layer



(b) Re-fashioned latent layer



(c) SVP creation process

Figure 8: An example of SVP sorting from [13] which uses the MNIST dataset. (a) Without any transformation, the innermost hidden layer, which has 64 nodes is shown here. (b) After sorting the 64 hidden layer nodes (rows) independently we get an image where the larger hidden layer values are on the left and smaller values are on the right. (c) Steps needed to convert a latent representation into SVPs that can be used to estimate dimension.

Table 2: AE architecture used for performing AEDE on the CICIDS2017 dataset.

Layer Type	**Nodes**	**Regularizer**	**Activation**
input layer	74	-	-
encoder layer 1	60	-	relu
dropout layer 1	Drop 1%	-	-
encoder layer 2	45	-	relu
dropout layer 2	Drop 1%	-	-
encoder layer 3	35	-	relu
dropout layer 3	Drop 1%	-	-
lambda layer	35	-	-
hidden layer	25	ℓ_1	sigmoid
decoder layer 1	35	-	relu
decoder layer 2	45	-	relu
decoder layer 3	60	-	relu
output layer	74	-	sigmoid

data for training, leaving the remaining 30% for validation. We removed all network identifying features such as IP addresses, ports, and protocols resulting in 74 features serving as input to our autoencoder. All features were normalized using min-max normalization based on the training data as described in Appendix A.3.1. We note that this operation increases the dataset dimension by one but does not affect our overall analysis as we are interested in comparing relative changes in dimension over time.

A hyperparameter search was performed across multiple λ values based on the resulting MSE of autoencoder reconstructions as depicted in Figure 9. A λ value of 0.0001 was chosen based on this search. The autoencoder architecture used is summarized in Table 2 where the number of layers and sizes were determined through ablation studies.

We can now estimate the dimensionality of the network while it is not under

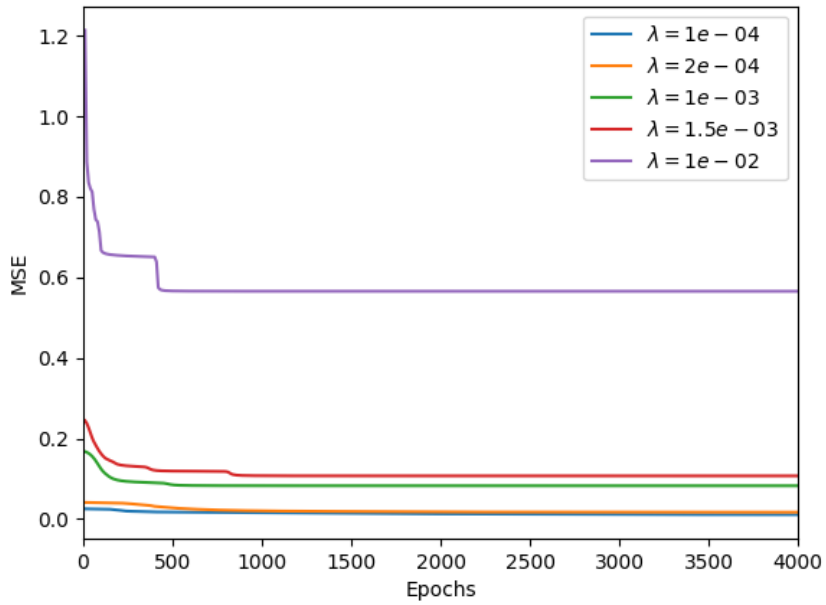


Figure 9: Choosing the value of the λ parameter to control the sparsity of the encoder's hidden layer was performed using the MSE of the autoencoder model. It can be seen that the lowest MSE correlates to a λ of 0.0001. Based on this, the λ chosen for the model was 0.0001.

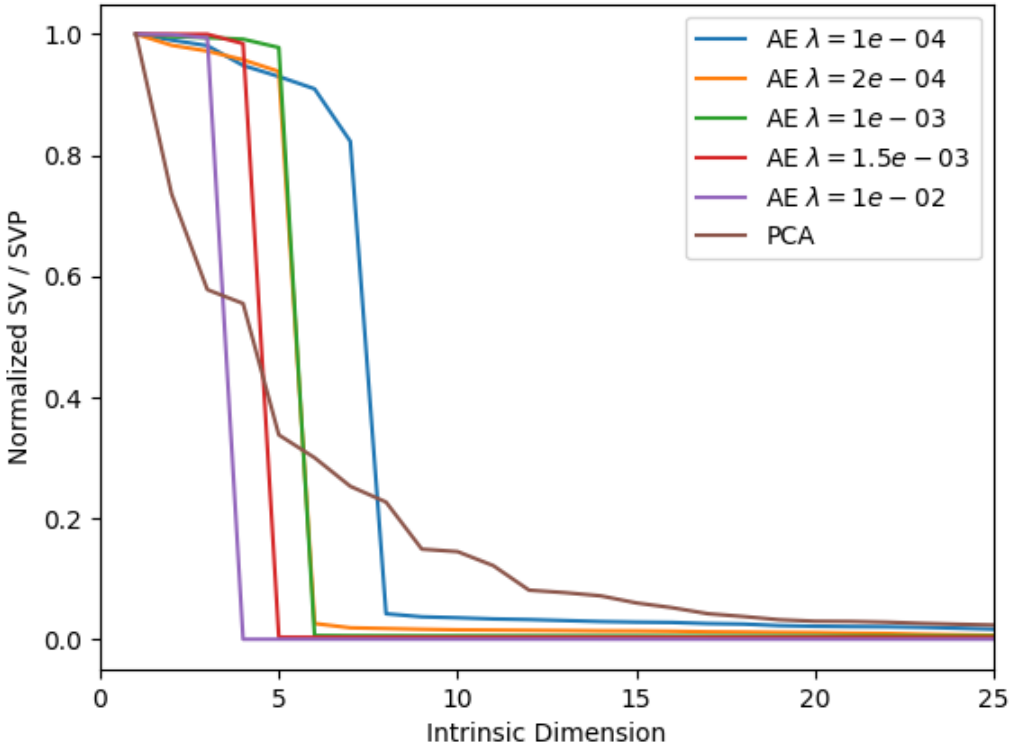


Figure 10: The scree plot generated using the singular value proxies for each autoencoder trained with varied values of λ to control the sparsity of the hidden layer of the encoder. Shown in blue, using the singular value proxies from the autoencoder trained with λ = 0.0001 assesses the dimensionality of the Monday benign network traffic data of the CICIDS2017 dataset to be seven based on the "knee" in the curve. For comparison, PCA is shown in brown and assesses the dimensionality of the data to be thirteen.

attack. These estimates across different values of λ are shown in Figure 10. Our chosen λ of 0.0001 estimates the dimensionality of the network to be seven, which is smaller than the linear PCA estimate of thirteen. For λ values which resulted in poor reconstruction performance compared to our chosen λ, one can see that the dimensionality is estimated to be less than seven.

After choosing the λ to use for the model, we considered where to set the singular value proxy threshold α. The task of finding the proper value of α is not

well-defined and relies on both empirical measures and judgement based on domain knowledge. Some insights can be gained by performing dimension estimates using several α of increasing magnitude as summarized in Figure 11. It is intuitive to consider that with α set to 1 we will estimate a dimensionality of 0, while an α of 0 will estimate a dimensionality equal to the number of nodes in the hidden layer of the encoder. In line with this, one can see that there is a range of α between 0.001 and 0.003 inclusive which appear to be thresholds that are not restrictive enough. Similarly, values of α greater than 0.006 appear too prohibitive. This leaves several values of α to look at closer. A value of 0.006 was chosen for α as its estimate of dimensionality aligned with the "knee" in the scree plot for the chosen model which can be seen by comparing Figures 10 and 11. It should be noted that choosing the optimal α is an open problem that could be explored further.

We used time intervals of 1, 5, and 10 minutes of network data to evaluate the performance of AEDE. From Table 3 and Figure 12 one can see that we obtain relatively consistent estimates across all time intervals on the benign validation data used. We now examine the results from different types of network attacks found in our dataset. The most success was found detecting DDoS attacks in Friday's scenario from the dataset as seen in Figure 13. Darker blues indicate a higher dimension estimate which shows that while under the attack, shown between two dashed lines, the network dimensionality increases. The remaining periods report relatively consistent dimensionality estimates. PCA also has a change in its dimensionality estimate during the attack period, however, there is a considerable amount of variability on either side of the attack when compared to AEDE. In addition, when provided with longer time intervals, PCA is unable to detect the attack period compared to AEDE.

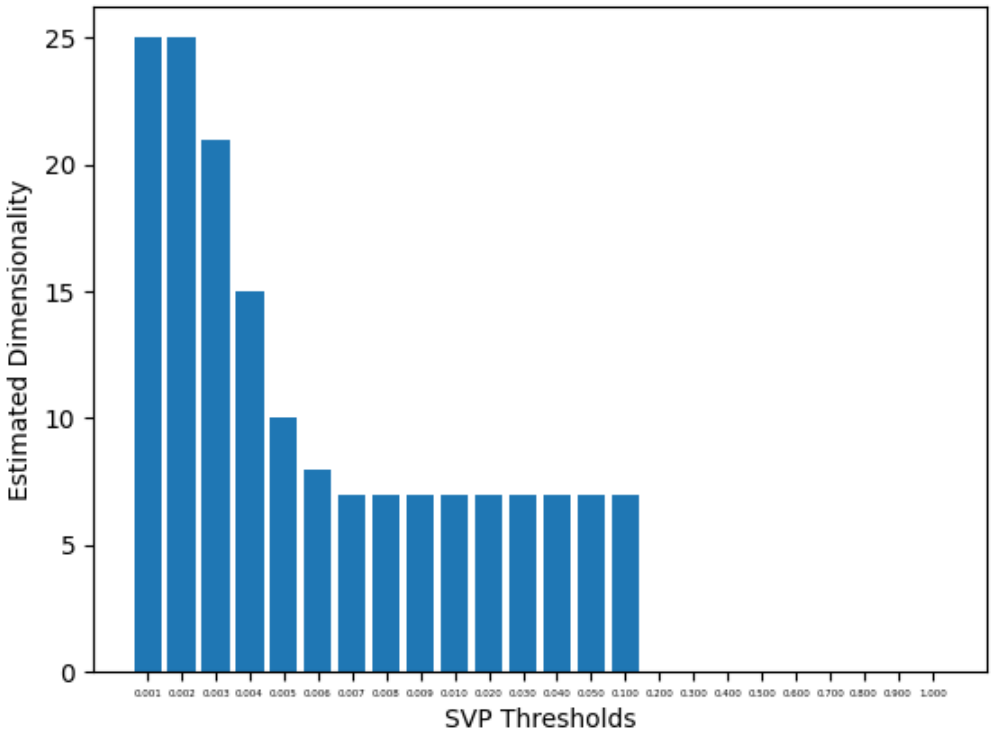


Figure 11: A study of increasing values of the singular value proxy threshold α was performed using the validation data. One can see an area which provides little filtering of singular value proxies until α = 0.004. Values of α greater than 0.006 appear to be too restrictive. A value of α = 0.006 was chosen for this work.

Table 3: Mean and standard deviation of dimension estimation on Monday's benign data from the CICIDS2017 dataset.

	1 Min.	**5 Min.**	**10 Min.**
PCA	18.83 ± 1.4	19.68 ± 0.2	19.77 ± 0.28
AE	7.24 ± 0.5	7.04 ± 0.2	7.08 ± 0.28

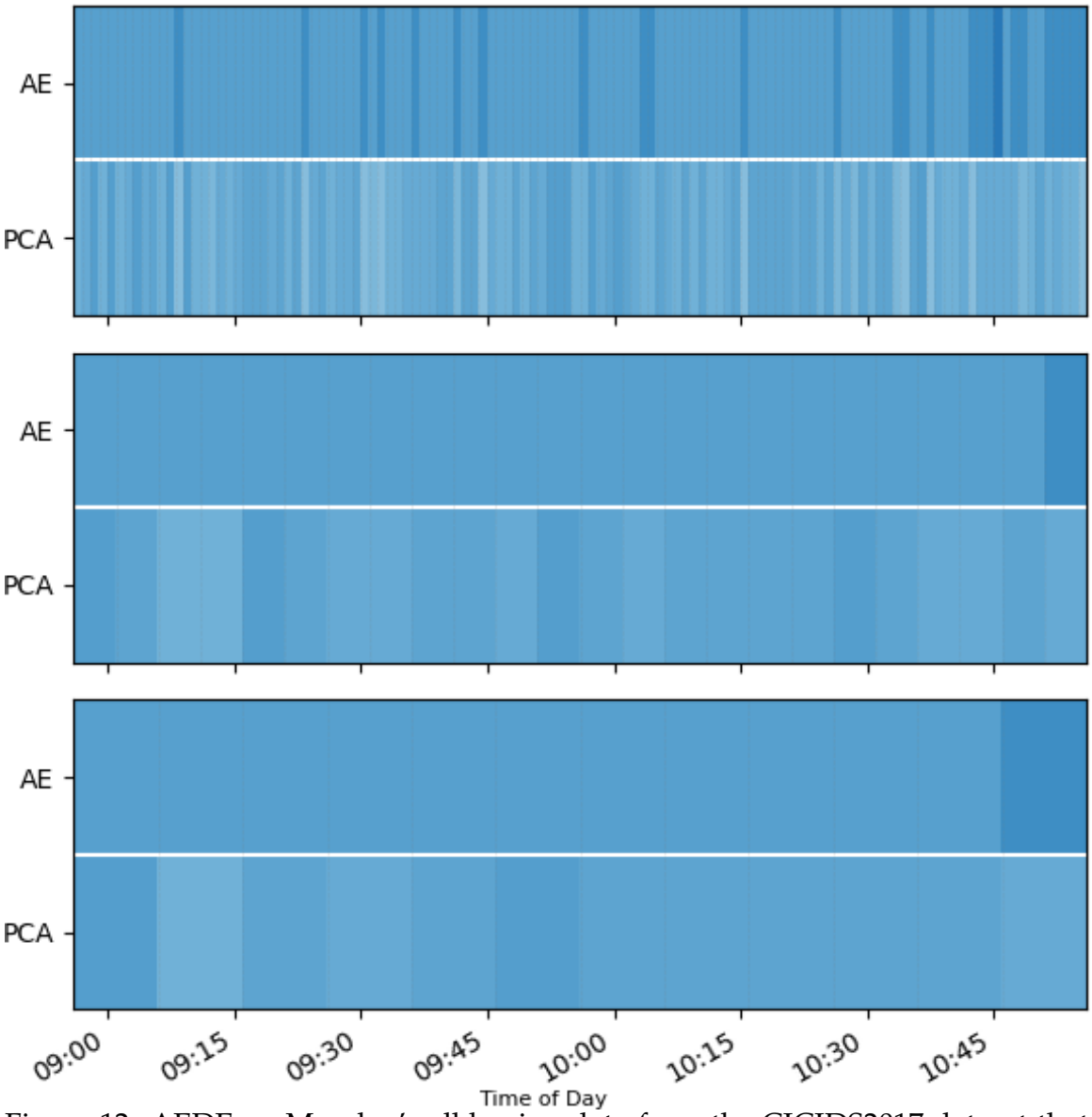


Figure 12: AEDE on Monday's all benign data from the CICIDS2017 dataset that visualizes the summary data reported in Table 3. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals. One can see that the dimensionality is estimated consistently for all time periods with the 5 and 10 minute periods yielding slightly better results.

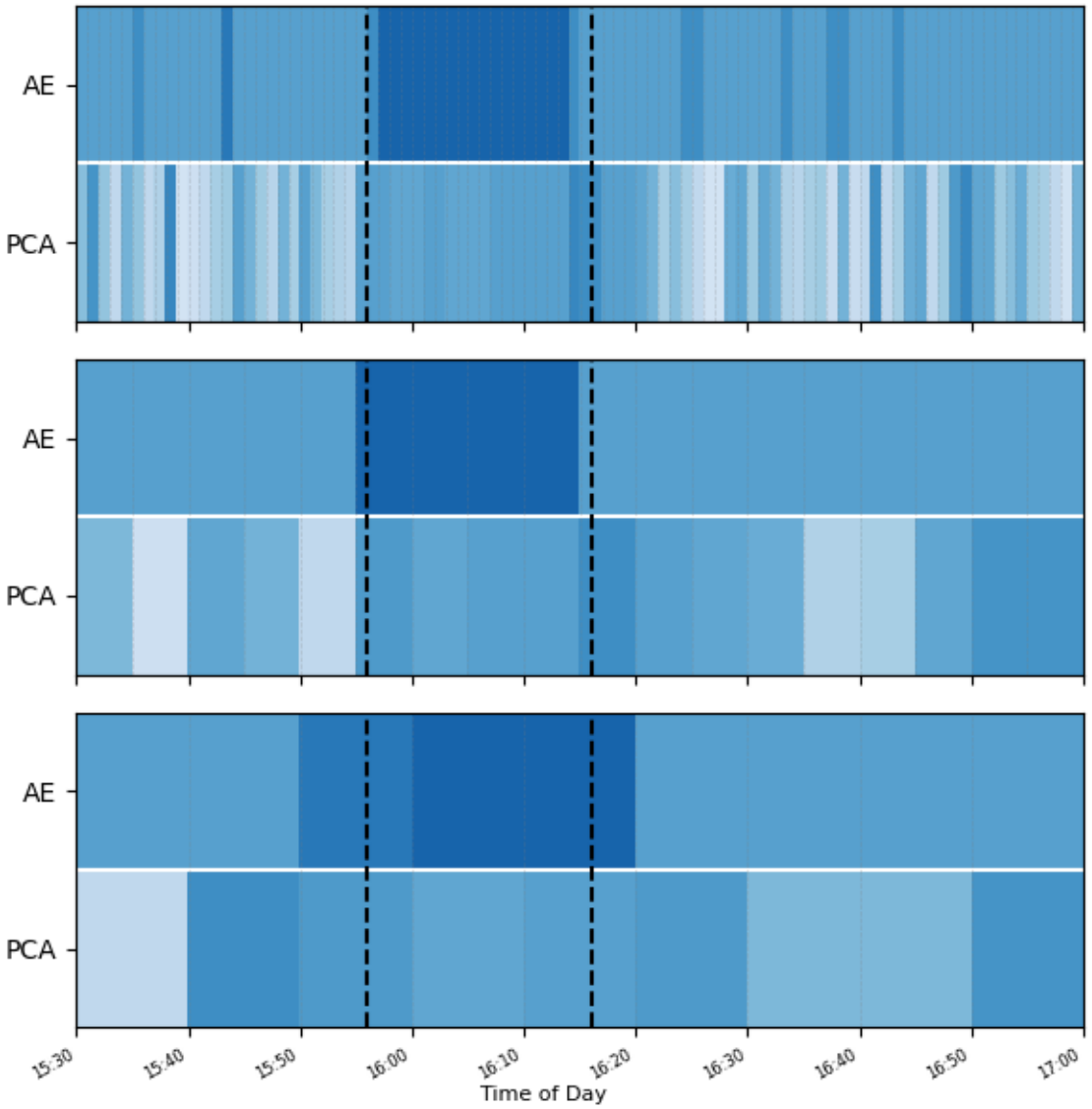


Figure 13: Dimension estimates of the Friday network traffic flows in the CICIDS2017 dataset. The AE used was trained for 4,000 epochs with $\lambda = 0.0001$ to control sparsity of the innermost hidden layer of the encoder. A SVP threshold of $\alpha = 0.006$ was used when estimating the dimensions. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals. Darker colors of blue indicate a higher estimate of network dimensionality. The time period between the two black dotted lines indicates when the network was under a Distributed Denial of Service (DDoS) attack. The autoencoder dimension estimate clearly shows an increase in the dimension of the network during the attack period. PCA shows a similar detection, however, with considerably more noise during benign periods and it is unable to handle larger time intervals such as the 10 minute interval.

This detection of large attacks held true for the Wednesday denial of service (DoS) attacks as depicted in Figure 14. However, for other more subtle attacks such as port scans and web attacks as depicted in Figures 15 through 18, AEDE was unable to perform well at providing a clear indicator of an attack taking place.

There are several takeaways that we can note from these findings related to using AEDE for network intrusion detection. First, its strength in this area is at detecting attacks which cause large network disruptions across the entire network such as DoS and DDoS attacks. With more subtle attacks, or attacks targeted at a specific machine, it is not well-suited. Additionally, given that this technique analyzes the network as a whole, it provides no indicator of which network devices are under attack or contributing to the attack conditions. For these reasons, we sought to continue researching this area and began exploring RDAs as discussed in Section 3.4 and ultimately began looking at autoencoder feature residuals which are discussed in Chapters 5 and 6.

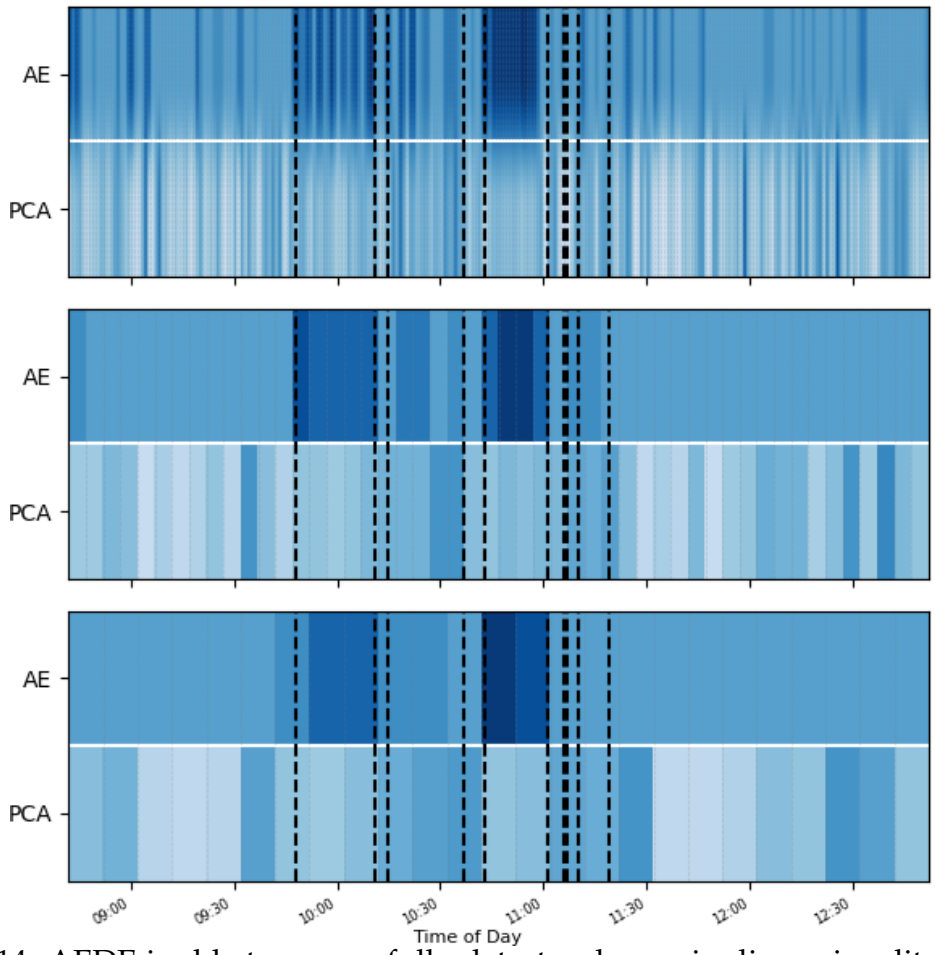


Figure 14: AEDE is able to successfully detect a change in dimensionality during multiple different denial of service attacks that occur in Wednesday's data from the CICIDS2017 dataset. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals. One can observe that AEDE yields more definitive results compared to PCA.

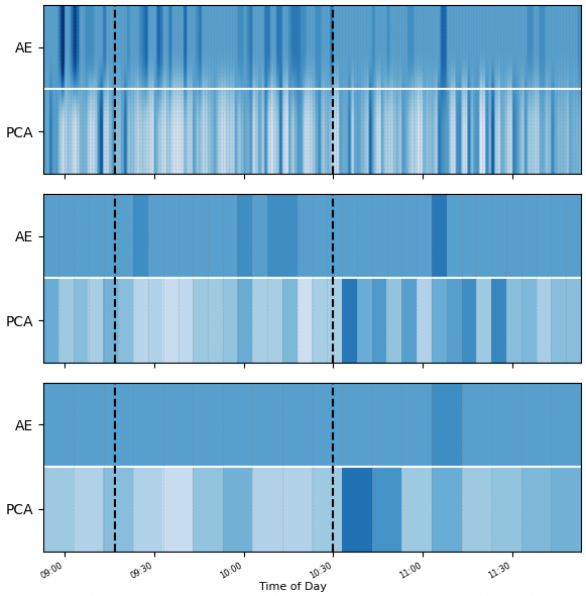


Figure 15: Both AEDE and PCA struggle to consistently detect changes in dimensionality during brute force attacks present in Tuesday's data from the CICIDS2017 dataset. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals.

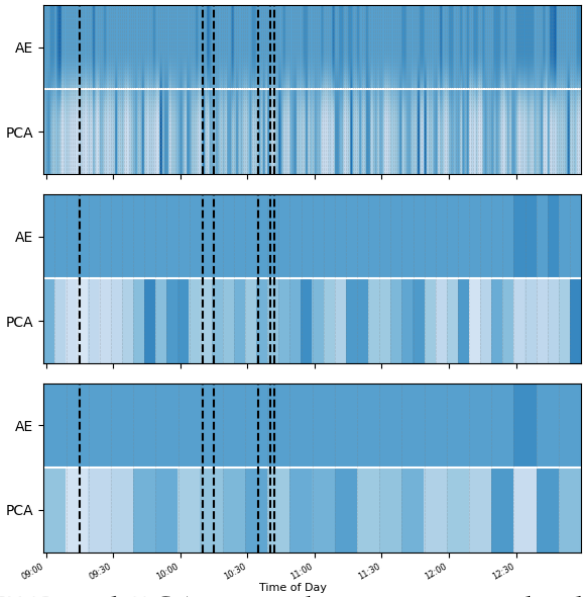


Figure 16: Both AEDE and PCA struggle to consistently detect changes in dimensionality during web attacks present in Thursday's data from the CICIDS2017 dataset. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals.

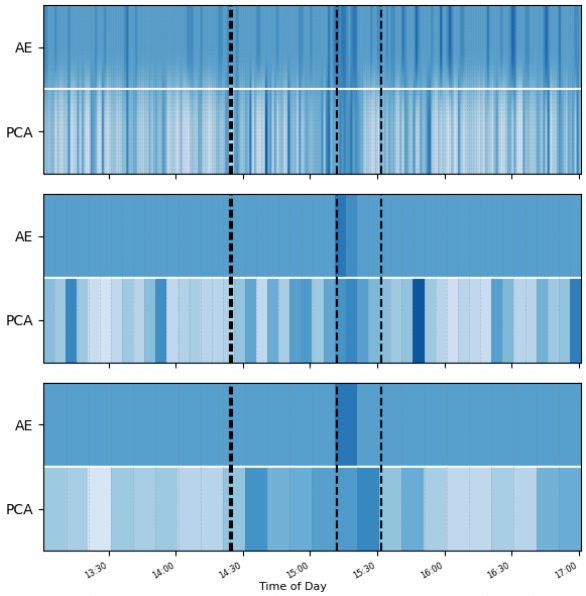


Figure 17: Both AEDE and PCA struggle to consistently detect changes in dimensionality during a heartbleed attack present in Wednesday's data from the CICIDS2017 dataset. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals.

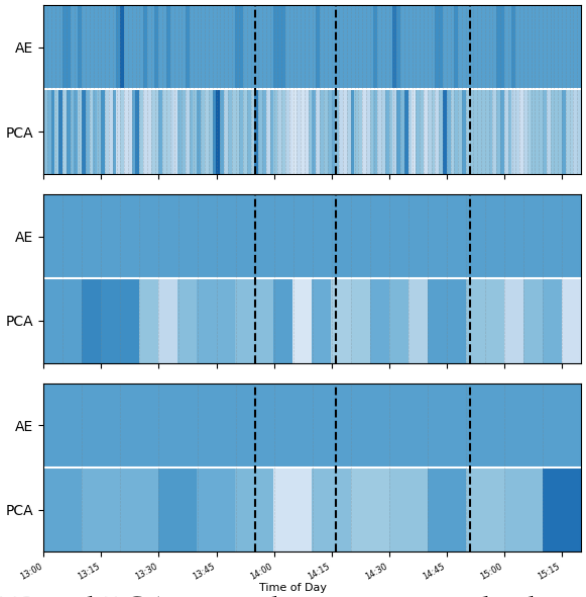


Figure 18: Both AEDE and PCA struggle to consistently detect changes in dimensionality during a port scan present in Friday's data from the CICIDS2017 dataset. The top plot estimates dimension at 1 minute intervals, the middle plot at 5 minute intervals, and the bottom plot in 10 minute intervals.

5 Autoencoder Feature Residuals

As noted in Chapter 4, after exploring AEDE we found that it struggled to generalize for network attacks with small footprints. Attacks such as advanced persistent threats (APT) and emerging zero-day exploits would likely be undetected by techniques that are unable to classify individual network flows as attack or benign. For this reason, inspired by the work of Zhou and Paffenroth [114, 78], we sought a technique that is able to differentiate normal samples from anomalies in a dataset. As described next, we developed the creation of novel feature sets using autoencoder feature residuals (AEFR) to meet this need. Here we define AEFR in a general way and reserve their application to NID for Chapter 6 where we review its application through exhaustive empirical analysis.

5.1 Defining Autoencoder Feature Residuals

At its core, AEFR provide unique feature sets for downstream methods. While we focus on classifiers in our work, the usage of these features need not be restricted to that downstream task. We start our definition by utilizing the structure of the autoencoder as explained in Section 2.6. We define X as some original feature set with k features. Providing X as input to a feed forward autoencoder we obtain the autoencoder reconstructions of X which we define as L in equation 35

$$L^k = D(E(X^k)) \tag{35}$$

where E is the encoder, D is the decoder, and L^k is the k-dimensional reconstruction of X produced as output by the autoencoder.

We then produce the autoencoder feature residuals by performing an element-

Table 4: The feature sets made possible using autoencoder feature residuals starting with an original set of k features we refer to as X. The autoencoder reconstruction of these features is referred to as L. The autoencoder feature residuals are referred to as S. While we focus solely on using all k elements from X, L, and S one could also create feature sets with a subset of the features as well.

Feature Vector	Number of Features
X	k
L	k
S	k
$\overrightarrow{XL}$	$2k$
$\overrightarrow{XS}$	$2k$
$\overrightarrow{LS}$	$2k$
$\overrightarrow{XLS}$	$3k$

wise subtraction of L from X as shown in equation 36

$$S^k = X^k - L^k \tag{36}$$

where S^k are the k autoencoder feature residuals.

At this point, one can create new feature sets by utilizing X^k, L^k, and S^k in combinations as outlined in Table 4. We note that we concentrate our research on utilizing all k elements from X, L, and S when creating feature sets, however, one could utilize subsets from these as well.

5.2 Autoencoder Feature Residuals Training Procedure

Having outlined the mechanical definition of AEFR and their generation, we note that the produced feature combinations outlined in Table 4 likely hold little value compared to X if not paired with a specific training procedure which we discuss in this section. We start with the intuition that an autoencoder will have an easier time

reconstructing samples that are the same or similar to those used during its training. The mirror to this statement is that the autoencoder will have a harder time reconstructing samples that are different from those seen during training. During inference on new samples, this results in the reconstruction error being higher for samples that differ from those seen during training. This idea is the basis of many anomaly detection techniques that use autoencoder residuals in an aggregate manner [8, 69, 79, 20].

Shown in Figure 19, we incorporate this idea into our training process by only training using samples from one class, which we will refer to as the normal class, and denote its samples as X_n^k. In training the autoencoder on only samples of this type, we aim to have an autoencoder that has a robust overall representation of what characterizes a sample as normal. We refer to the anomalous class using the subscript d such that we can represent samples of that class as X_d^k. With this in mind, when training an autoencoder to generate features using AEFR, we seek to minimize equation 37

$$\min_{\theta} \ell(X_n^k - D(E(X_n^k))) \tag{37}$$

where θ are the autoencoder weights and biases we are learning and ℓ is our loss function to calculate the difference between our input and reconstruction of normal samples. During inference, we attempt to reconstruct both normal and anomalous samples and represent each case using equations 38 and 39

$$L_n^k = D(E(X_n^k)) \tag{38}$$

$$L_d^k = D(E(X_d^k)) \tag{39}$$

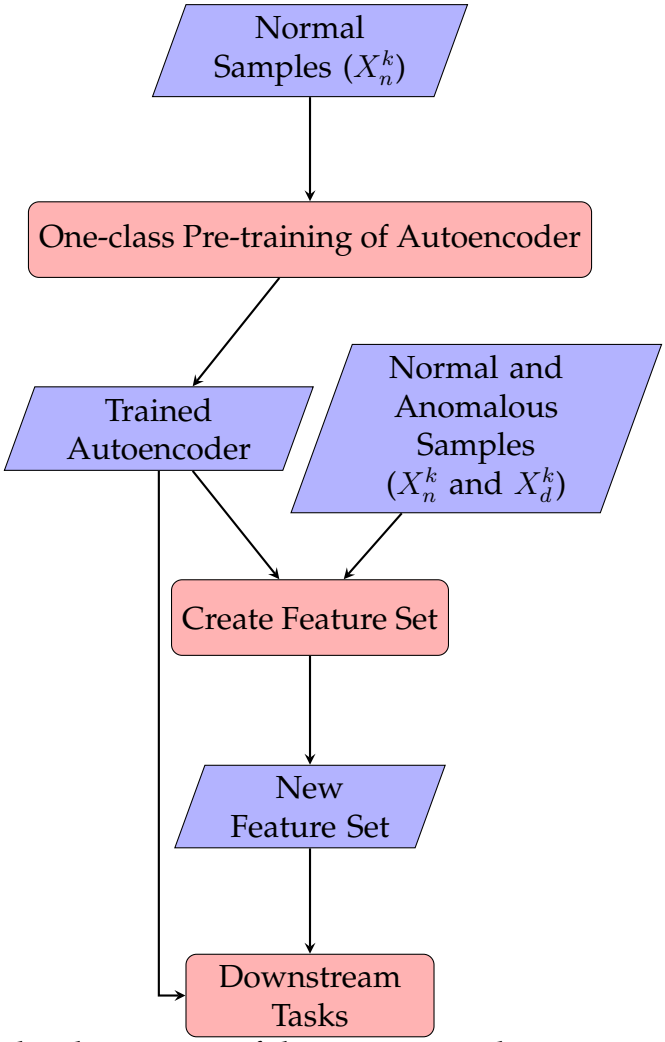


Figure 19: A high level overview of the process used to generate and use autoencoder feature residuals. First, a one-class pre-training step is performed to produce an autoencoder using only normal samples of data which we refer to as X_n^k. The trained autoencoder is then provided with both normal and anomalous data samples to generate feature sets using the original features, autoencoder reconstructions, and autoencoder feature residuals. We refer to anomalous samples as X_d^k. The feature sets and trained autoencoder are then available for use in downstream tasks.

which allows us to then compute the AEFR represented by equations 40 and 41.

$$S_n^k = X_n^k - L_n^k \tag{40}$$

$$S_d^k = X_d^k - L_d^k \tag{41}$$

Note that when we refer to a mixture of normal and anomalous samples we continue using the notation from equations 35 and 36.

5.3 Properties of Autoencoder Feature Residuals

5.3.1 General Properties

The general properties that relate to AEFR are relatively straightforward with most of them having been outlined in Sections 5.1 and 5.2. For completeness we want to explicitly state that when working with AEFR and the feature sets that can be generated with them, there is no new data being created. In other words, all the data that is used was already present in our initial X^k and our processing through the autoencoder to generate L^k and S^k is a restructuring of this data. To compliment this idea we can express this using equations 42, 43, and 44.

$$X^k = L^k + S^k \tag{42}$$

$$X_n^k = L_n^k + S_n^k \tag{43}$$

$$X_d^k = L_d^k + S_d^k \tag{44}$$

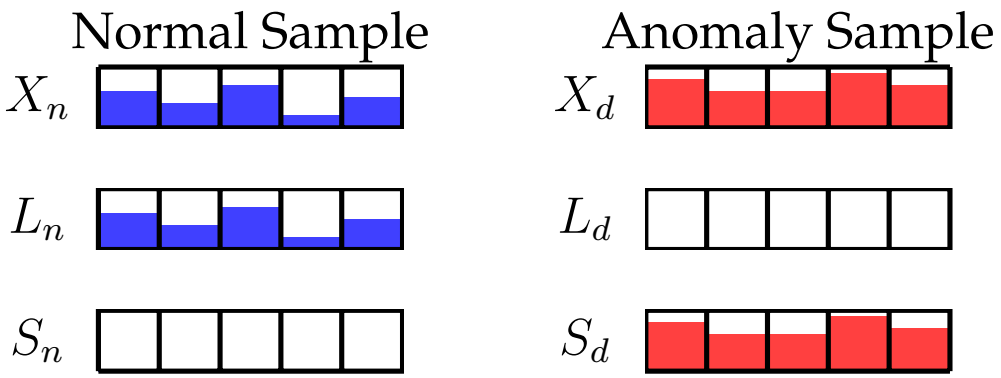


Figure 20: An example of the optimal result one could hope to achieve when working with autoencoder feature residuals. Here we have a normal sample on the left and a pure anomalous sample on the right each with five features. The color bars represent the value of each feature. The top row shows the original features X, which are put through an autoencoder to produce L in the middle row. In this ideal scenario, the autoencoder has perfectly reconstructed X_n, making $L_n = X_n$ and $S_n = 0$, while at the same time not being able to reconstruct any portion of the anomalous sample X_d. This leaves $L_d = 0$ and $S_d = X_d$. Having an autoencoder that behaved in this manner on all samples of X, one could use either L or S as optimal features for downstream tasking such as classification or revert to using a summary metric as is often done in anomaly detection. In reality, such a result is unattainable as there will generally be some error in reconstruction of normal samples and some portion of anomalies that can be reconstructed.

Moreover, in the absence of the ideal conditions discussed in Section 5.3.2, using only L^k or S^k for features results in strictly having removed data from X^k.

5.3.2 Ideal Conditions

When working with AEFR one can identify a set of ideal conditions that would prompt for using either L^k or S^k as features for optimal anomaly detection. This situation would occur if one had an autoencoder that could reconstruct all normal samples without error, while at the same time, being unable to reconstruct any portion of the anomalous samples. Depicted in Figure 20, we can express this using equations 45 through 48.

$$L_n^k = X_n^k \tag{45}$$

$$S_n^k = 0 \tag{46}$$

$$L_d^k = 0 \tag{47}$$

$$S_d^k = X_d^k \tag{48}$$

With these conditions being met, one could use L^k or S^k as optimal inputs to downstream anomaly detection tasks. Meeting these conditions is unlikely for most real-world datasets, however, we note them here as what to strive for when working with this technique.

5.3.3 Failure Conditions

There are several general conditions to note that can cause this technique to be less effective than desired. First, the anomaly detection capabilities of the features generated with AEFR is affected by the quality of the normal training data used for the autoencoder. If it is not trained using adequately difficult normal samples that are representative of realistic conditions during inference, it is likely to produce S_n^k with higher values, which is undesirable.

Similarly, if inference is performed on problems that are prone to frequent concept drift, then the autoencoder may struggle to reconstruct emerging benign samples. As a way to mitigate this, one could perform continuous training of the autoencoder to account for changes in the target environment.

The generation of AEFR as formulated here consists of using a feed-forward autoencoder which does not use any form of memory. If the features of a sample could be either normal or anomalous given the context surrounding the sample, the actual AEFR will not be able to account for these differences. There are several ways to mitigate this such as re-formulating the input to the autoencoder to consist of data from multiple individual samples. Another alternative is to use a downstream method that can account for the context of the sample such as an RNN or LSTM classifier.

5.4 Comparison to Aggregate Residuals

It is common to see anomaly detection performed using an autoencoder trained on normal data after which, during inference, anomaly detection is based on an aggregate error metric of the reconstruction along with a threshold [8, 69, 79, 20]. As the usage of AEFR is much less prevalent, it begs a look into why we would expect AEFR to have an advantage over such techniques.

An example of a common aggregate residual used is MSE as outlined in Appendix A.3.2. During inference the MSE of a sample is calculated and compared to a generally user-defined threshold. If the sample MSE, commonly referred to as anomaly score, is higher than a threshold, it is considered an anomaly. We start our discussion by noting that all the failure conditions for AEFR discussed in Section 5.3.3 also apply when using an aggregate residual. However, the situation is much worse for techniques using aggregate residuals as the anomaly score has lost valuable data that AEFR preserve. This can be shown through a synthetic example depicted in Figure 21 where we have S^k for two samples with $k = 5$. The reconstruction error for the samples are feature-wise mirror images of each other. One

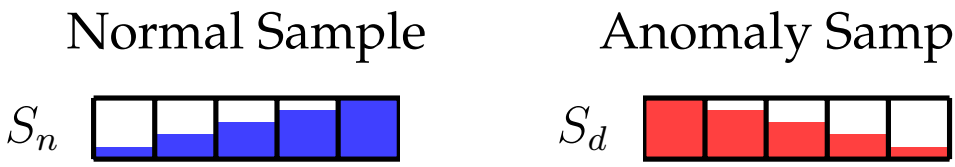


Figure 21: In this synthetic motivating example we show the disadvantages that can arise when using aggregate loss metrics for detecting anomalies. On the left hand side we show the AEFR for a normal sample, while the right hand side shows the AEFR for an anomaly. The error for both samples is equal, however, it is distributed differently among the features. An aggregate metric such as MSE would calculate the anomaly score the same for both samples, resulting in the normal sample being flagged as a false-positive. Providing the AEFR to a downstream classifier provides the classifier with the opportunity to differentiate the two samples.

sample is normal while the other is anomalous, however, the MSE calculated for each reconstruction are equal. Relying on an aggregate residual metric is unable to determine a difference between the two samples resulting in one being misclassified as an anomaly. The data necessary to do this, however, is preserved when using AEFR since we can see that there are differences in which *individual features* have high residuals for the normal and anomalous sample.

As another motivating example, one can see Figure 22, which shows another synthetic example of S^k for a normal and anomalous sample. In this situation, however, we have minor residuals for all features of the normal sample, while the anomalous sample is characterized by a single large anomalous feature that the autoencoder could not reconstruct. Again in this situation, using aggregate residuals is not adequate as it has removed the details necessary to properly differentiate the normal sample from the anomalous sample.

Another shortcoming related to working with aggregate residuals that we need not deal with (unless desired) is the usage of a threshold to detect anomalies. This threshold may be different across domains and imposes an additional hyperpa-

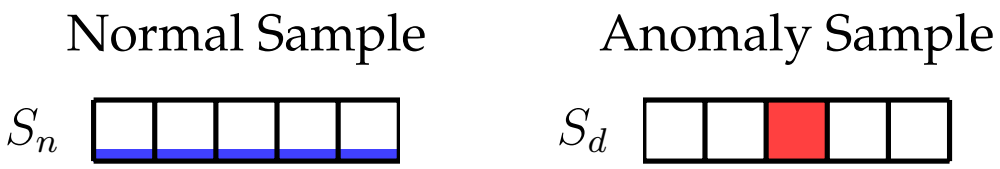


Figure 22: In this synthetic motivating example we show the disadvantages that can arise when using aggregate loss metrics for detecting anomalies. On the left hand side we show the AEFR for a normal sample, while the right hand side shows the AEFR for an anomaly. The normal sample has a loss of 0.1 across all features for a total of 0.5. The anomaly has a loss of zero for four of its features, while the autoencoder could not reconstruct feature three which has a loss of 0.5. Here, an aggregate residual metric would consider both of these anomalies as it does not account for the structure of the errors. Using AEFR allows downstream tasks to take advantage of this structure for performance improvements.

rameter to tune, or the exploration of normalization procedures to apply to the metric prior to thresholding. While we have this option when working with AEFR, it is not strictly necessary given the granularity of information that is being maintained. Additional options, such as using AEFR along with a downstream neural network classifier remain at our disposal.

5.5 Synthetic Examples

Prior to investigating the application of AEFR to real-world datasets, we first explored its usage on synthetic datasets that we constructed. With these datasets, we defined a pattern that we considered to be normal and then injected random anomalies of a specific value. We consider these synthetic samples relatively easy, so if we were unable to detect anomalies in this situation, it would be likely that the technique would be unable to translate over to NID datasets.

In the first scenario, we defined our normal samples to be those containing features with a value of one while injecting random values of two. After training an autoencoder on normal samples, our expectation would be for L^k to contain mostly

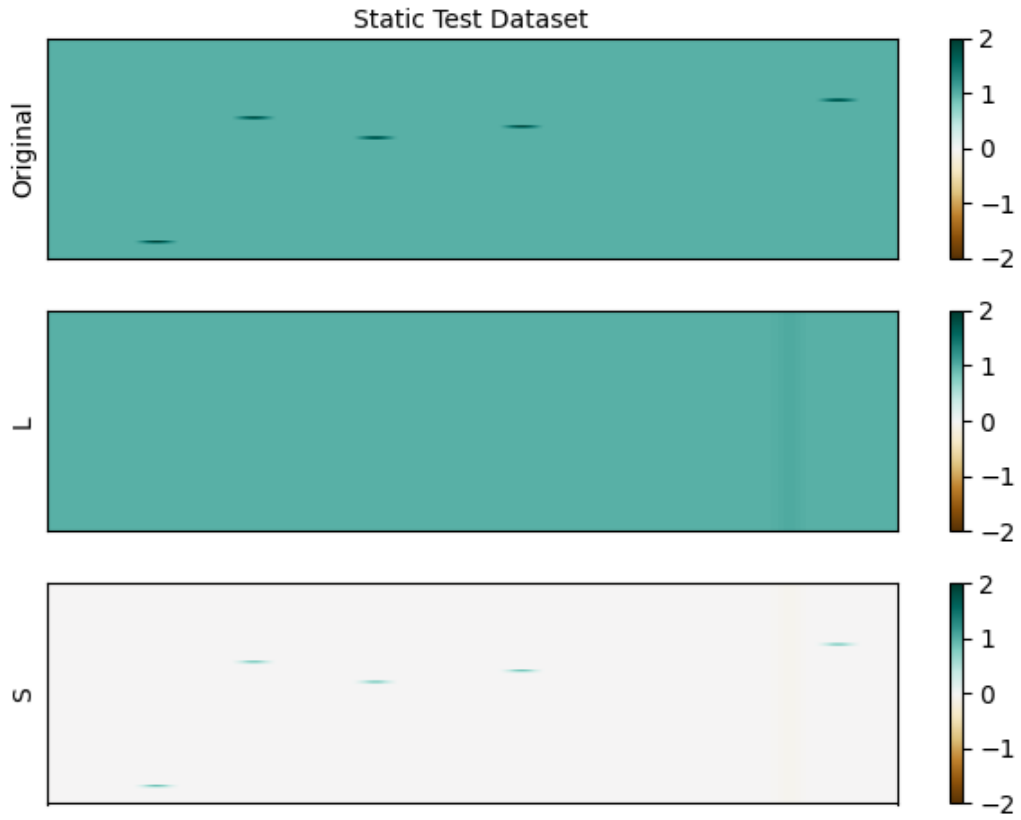


Figure 23: A synthetic example where normal samples contain features with all their values set to one. An autoencoder was trained on normal samples only. Anomalous entries contain features with a value of two. One can see that after training on normal samples L^k is dominated by values of one while S^k is sparse and contains the anomalies.

values of one, while S^k would be sparse except for areas where the anomalies were injected. This can be seen in Figure 23, where both of these expectations are met.

Figures 24 and 25 show two additional examples of increasing complexity which were both met with success. While there are some subtle artifacts that can be seen in all of these examples in both L^k and S^k, it remains clear that the training procedure for generating AEFR is able to successfully break X^k into an L^k that contains the easy to reconstruct portions of a sample while S^k contains the more difficult anomalies. We take this moment to acknowledge that we liken our L^k and S^k to

Figure 24: A synthetic example where normal samples are represented by rows with either values of one or values of two. An autoencoder was trained on normal samples only. Anomalous entries contain features with a value of five. One can see that after training on normal samples, L^k is dominated by values of all ones or all twos, while S^k is sparse and contains the anomalies.

the L_{rda} and S_{rda} covered in Section 3.4. Given this success, we moved to applying the use of AEFR for NID in order to detect network attacks as discussed in detail in Chapter 6.

Figure 25: A synthetic example where values of the features for normal samples are generated using a sin function. An autoencoder was trained on normal samples only. Anomalous entries contain features with a value of five. One can see that after training on normal samples, L^k is dominated by rows constructed by the sin function, while S^k is sparse and contains the anomalies.

6 Autoencoder Feature Residuals Applied to Network Intrusion Detection

In this chapter we cover the application of the concepts from Chapter 5 to NID. We note that our goal in this endeavor is to empirically determine general applicability of the feature sets produced by AEFR to the area of NID. In doing so, we cover a wide range of types of downstream classifiers, across a multitude of NID datasets, and using several encodings of the data. This breadth, however, likely came at a cost for leaving performance on the table by not optimizing our architectures and hyperparameters for individual NID scenarios. In this regard, we actually believe this is a more practical approach for moving research into actual NID settings due to the volatility and dynamic scenarios that must be covered in this area, which we touch upon in Section 6.3.

6.1 Methodology Overview

Here we simplify the notation from Chapter 5 to be specific to the NID problem domain as well as outline our overall methodology, deferring details to the following sections. In this domain, we consider the normal samples to be benign network traffic and the anomalies to be attack network traffic. In line with this, we adjust the notation used for our original set of input features to be X, X_b, and X_a where we have dropped the k and the subscript b represents only benign network samples while the subscript a represents only attack network samples, while no subscript represents a mixture of both benign and attack network samples. Similarly, we have equations 49, 50, and 51 to represent autoencoder reconstructions on network data,

$$L = D(E(X)) \tag{49}$$

$$L_b = D(E(X_b)) \tag{50}$$

$$L_a = D(E(X_a)) \tag{51}$$

again dropping the k and using the subscripts b and a. Finally, we have the same adjustments made to represent the AEFR as shown in equations 52, 53, and 54.

$$S = X - L \tag{52}$$

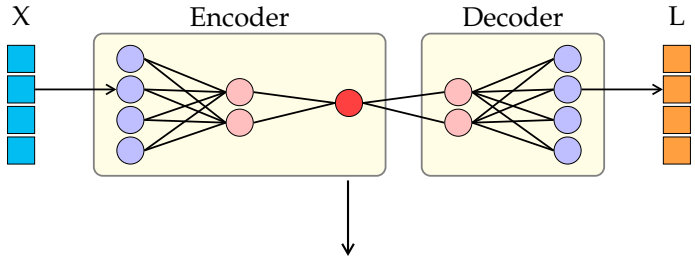
$$S_b = X_b - L_b \tag{53}$$

$$S_a = X_a - L_a \tag{54}$$

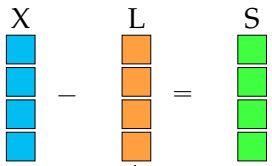
For our application of AEFR, we pair the generation of the feature sets using AEFR along with a large selection of classifiers which include traditional machine learning models, neural networks of various architectures, as well as unsupervised methods executed in a one-class manner. Our overall general process, which follows from that outlined in Chapter 5, is depicted in Figure 26.

We begin by training an autoencoder using only benign NetFlow samples. In doing so, we are making an assumption that after training, the autoencoder will be able to easily reconstruct unseen benign NetFlow samples, while struggling with attack NetFlow samples. After training is complete we are able to generate L and

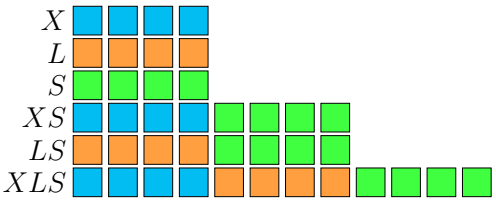
1. During training we use only benign samples to train our autoencoder. Inference is performed on benign and attack data.



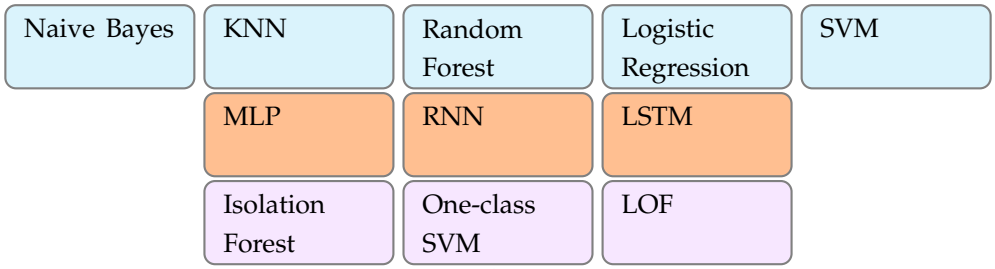
2. Create autoencoder feature residuals: $X - L = S$



3. Use X, L, and S to create novel feature combinations.



4. During training only benign samples used to train classifiers. Inference is performed on benign and attack samples.



5. During inference our trained autoencoder and classifier are used together to classify network traffic as benign or attack.

Figure 26: The overall process for constructing and using feature sets that take advantage of autoencoder feature residuals as we applied it to NID. In blue we show the various classical machine learning models we used for downstream classification. In orange we show the various neural network architectures we used for downstream classification. In purple we show the various one-class classifiers we used for downstream classification.

S using both benign and attack NetFlow samples. With X, L, and S in hand we construct our feature sets and use each of them to train a separate downstream classifier. We are then able to assess the technique using a holdout test dataset using both the trained autoencoder and classifier.

6.2 Performance Metrics

We collected a number of metrics throughout our research in order to assess the performance of our method on classifying network attacks. In general, we most commonly report the f1-Score as it remains more suitable compared to accuracy for problems with a heavy class imbalance such as NID. Additionally, we provide an analysis of the false alarm rate as it carries significant real-world implications in the area of NID as discussed in Section 6.8. We define the metrics utilized as part of the research outlined in this dissertation in Appendix A.2.

6.3 Encoding Network Intrusion Detection Data

We discussed the difficult intrinsic properties of network data in Section 2.1.3. Here we discuss one final complexity and outline how we accounted for these difficulties in our research. Namely, one issue that complicates both research and the practical application of machine learning to cybersecurity is that there is no standard set of features to use for the task [90]. This issue stems from various points in the process of applying machine learning to cybersecurity. First, network administrators and researchers must determine which attributes to actually collect from their network infrastructure. If there are enough resources to collect full packet captures then there is no major issue, however, this is rarely the case. This results in the collection of NetFlow data and likely only a subset of all the available fea-

tures of NetFlow data.

This affects downstream consumers of the data attempting to perform machine learning as each dataset has different features derived from network traffic [90]. Looking at Table 26 in Appendix A.4, one can see that aside from related datasets, none of the commonly used benchmark datasets deliver the same feature set.

For this reason we use an encoding that consists of the simple features shown in Table 5 that can be commonly found across a wide variety of datasets. In general we focus on features derived from bytes, packets, protocol, and ports which are all generally available in benchmark datasets. Additionally, we conducted studies summarized in Table 6 to determine if features derived from bytes and packets were helpful in performing classification prior to including them in our encoding.

Though not available at the start of our research, we also later utilized a set of datasets from Sarhan [91] and described in Appendix A.4, which encodes the data using standard NetFlow v9 features which we refer to as the NFV2 encoding.

6.4 Determining Autoencoder Architecture

In early work we performed multiple ablation studies to determine the autoencoder architecture to use for generating AEFR. There are several notable observations from these studies related to our choice of autoencoder architecture. We found that using the ReLU activation function between hidden layers provided the best results in terms of reconstruction loss across a wide variety of NID datasets. Additionally, in early work we performed ablations across both the number of layers in the encoder and decoder combined with different levels of compression in the bottleneck layer. Figure 27 shows results across the extremes of the ablation study for the CTU-13 Scenario 6 dataset where we paired using AEFR with a

Table 5: The encoding of NetFlow features used for the majority of this research. We utilized features based on bytes, packets, protocol, and ports which are all generally available across benchmark datasets. Details regarding the one hot encodings can be found in Appendix A.5.1 and A.5.2.

Feature (Type)	**Description**
communication_type (OHE)	L4_[SRC|DST]_PORT
protocol (OHE)	PROTOCOL
destination_bytes_per_second (Float)	Natural log
destination_packets (Float)	Natural log
destination_packets_per_second (Float)	Natural log
duration (Float)	Duration in seconds
packets_per_second (Float)	Natural log
source_bytes_per_second (Float)	Natural log
source_packets (Float)	Natural log
source_packets_per_second (Float)	Natural log
total_bytes (Float)	Natural log
total_bytes_per_second (Float)	Natural log
total_destination_bytes (Float)	Natural log
total_packets (Float)	Natural log
total_source_bytes (Float)	Natural log
label (Binary)	1 = attack

Table 6: Summary metrics from a study conducted to determine which features to use in our encoding of NetFlow data. Our baseline features consisted of source and destination bytes and packets as features. We then independently added either a new set of columns or new columns with a transform applied to them. We used the mean f1-score using a random forest classifier trained on 34 network scenarios as our metric. In general, we used all features that improved the f1-score in our final encoding.

Feature	**Mean F1-Score**
Baseline	0.765
Bytes Per Second	0.783
Packets Per Second	0.775
Log Applied to Bytes Columns	0.879
Log Applied to Packets Columns	0.777
Log Applied to Bytes Per Second Columns	0.850
Log Applied to Packets Per Second Columns	0.783

random forest classifier, though its results are representative of the trends across most datasets used in the study. We show additional details from these studies in Appendix A.7. We see more consistent results for the various inputs used as we increase the number of layers in our autoencoder and expand the bottleneck size. Based on these results, most of our autoencoders use an architecture consisting of six layers in the encoder and decoder and a bottleneck size of 12.

An additional concern with the autoencoder architecture is regarding the final output layer and what activation function to use. As we apply a min-max normalization to our data one may be tempted to apply a sigmoid function on the output layer, however, we found the results when combining this with AEFR to be disappointing. We found more success by applying no activation function to the output layer of our autoencoder. Table 7 provides the detailed information for the autoencoder architectures used in our early research pairing AEFR with classical machine learning algorithms. Table 8 provides the detailed information for the

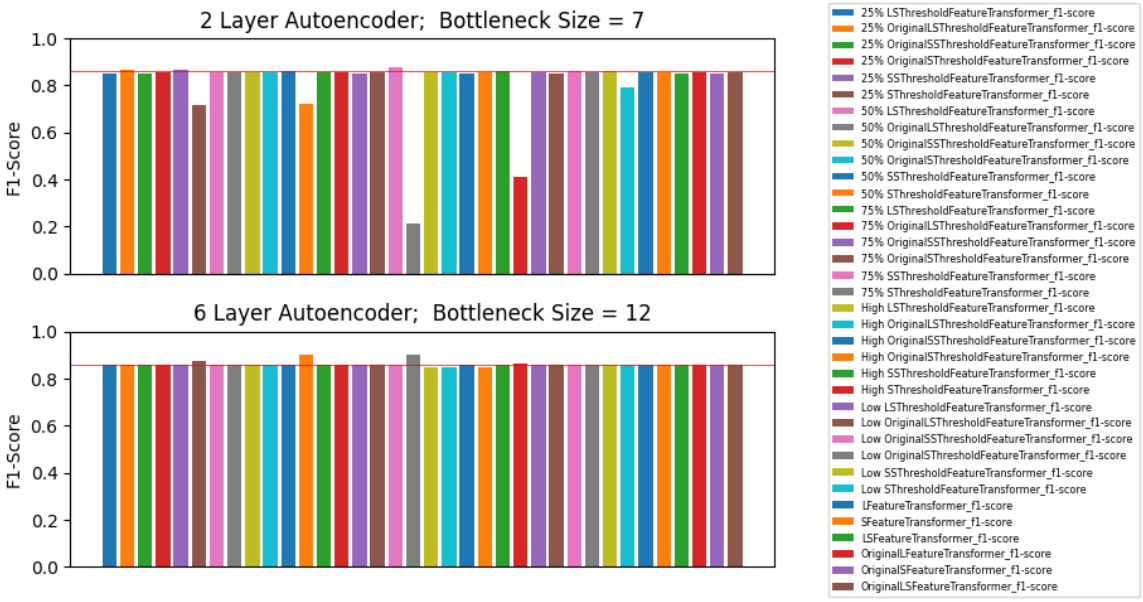


Figure 27: Snapshot of the extremes of an ablation study run on the CTU-13 Scenario 6 dataset. We paired AEFR inputs with a downstream random forest classifier to see the best results. In general, increasing the number of layers in the autoencoder while also increasing the bottleneck size provided more consistent results across all combinations of inputs used.

autoencoder architectures used throughout our remaining research for both our simple encoding and the NFV2 encoding of network data.

6.5 Training Autoencoder on Network Intrusion Data

When training our autoencoders for generating AEFR, as mentioned previously, we only provide benign network flow samples. Additionally, we use the Adadelta optimization function with the initial learning rate set to 1.0. MSE is used as the loss function during our training.

There are several other particulars worth discussing in this area. First, we initially trained our autoencoders using a batch size of 32. This was adequate for the NID datasets we used initially, however, we found using the same batch size on other NID datasets resulted in shrinking gradients and a collapse in training. To avoid this, we found that increasing the batch size to 128 prevented this situation from occurring, and we were able to maintain stable training.

Additionally, we found that we had to train our autoencoders for no less than 400 epochs in order to have a fully trained autoencoder, though we often train to 500 epochs when we approach new datasets in later work. While this seems like a trivial step, given the downstream usage of AEFR, without checking explicitly for convergence it is easy to miss an undertrained autoencoder. Shown in Figure 28, we see an example of an undertrained autoencoder and the resulting L and S. Here we see that L is a relatively constant value per feature. When using S in downstream tasks, it ends up performing as well as X as it is simply a shifted version of X. To avoid this confusion, it is imperative to have a fully trained autoencoder when generating AEFR.

Table 7: Architecture of the autoencoders used in our early research working with classic machine learning algorithms. The architectures below use a bottleneck size of 12. The number of nodes in each layer is adjusted slightly when a different bottleneck size is used. All autoencoders used the Adadelta optimization function with a learning rate of 1.0 and mean squared error for the loss function.

	2 Layer AE	**3 Layer AE**	**4 Layer AE**	**5 Layer AE**	**6 Layer AE**
Input Dimension	35	35	35	35	35
Encoder Layer 1	23	27	29	30	31
Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Encoder Layer 2	12	19	23	25	27
Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Encoder Layer 3	-	12	17	20	23
Activation	-	ReLU	ReLU	ReLU	ReLU
Encoder Layer 4	-	-	12	15	19
Activation	-	-	ReLU	ReLU	ReLU
Encoder Layer 5	-	-	-	12	15
Activation	-	-	-	ReLU	ReLU
Encoder Layer 6	-	-	-	-	12
Activation	-	-	-	-	ReLU
Decoder Layer 1	23	19	17	15	15
Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Decoder Layer 2	35	27	23	20	19
Activation	-	ReLU	ReLU	ReLU	ReLU
Decoder Layer 3	-	35	29	25	23
Activation	-	-	ReLU	ReLU	ReLU
Decoder Layer 4	-	-	35	30	27
Activation	-	-	-	ReLU	ReLU
Decoder Layer 5	-	-	-	35	31
Activation	-	-	-	-	ReLU
Decoder Layer 6	-	-	-	-	35

Table 8: Architectures of the autoencoders used with our most current research. Both autoencoders use the same general structure with 6 layers in the encoder and decoder and a bottleneck size of 12. The ReLU activation function is used for each layer except for the final output layer which does not use an activation function. The autoencoders use the Adadelta optimization function with a learning rate of 1.0 and mean squared error for the loss function. There is a slight difference in the number of nodes in each layer to account for the NFV2 dataset having 39 input features and the simple encoding having only 35.

	NFV2 AE	**Simple Encoding AE**
Input Dimension	39	35
Encoder Layer 1	34	31
Activation	ReLU	ReLU
Encoder Layer 2	29	27
Activation	ReLU	ReLU
Encoder Layer 3	24	23
Activation	ReLU	ReLU
Encoder Layer 4	19	19
Activation	ReLU	ReLU
Encoder Layer 5	14	15
Activation	ReLU	ReLU
Encoder Layer 6	12	12
Activation	ReLU	ReLU
Decoder Layer 1	14	15
Activation	ReLU	ReLU
Decoder Layer 2	19	19
Activation	ReLU	ReLU
Decoder Layer 3	24	23
Activation	ReLU	ReLU
Decoder Layer 4	29	27
Activation	ReLU	ReLU
Decoder Layer 5	34	31
Activation	ReLU	ReLU
Decoder Layer 6	39	35

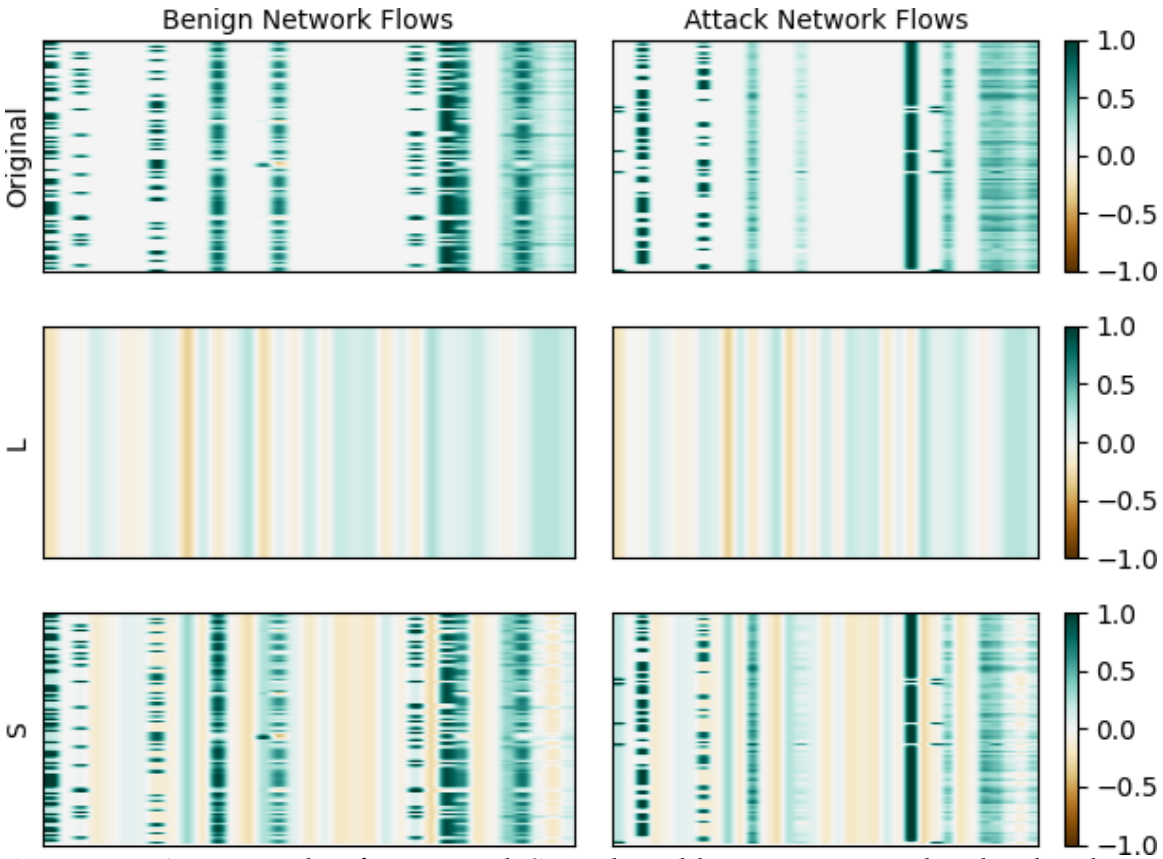


Figure 28: An example of X, L, and S produced by an autoencoder that has been undertrained. In this case, L is essentially a constant value and S is a shifted version of X. Without explicitly looking, one could mistake S as being effective for downstream tasks, however, the data provided to downstream tasks is closer to simply using X.

6.6 AEFR Effectiveness for NID

To assess the effectiveness of using AEFR for NID we paired the AEFR technique with a number of various types of downstream classifiers. Additionally, we utilized both our simple encoding and the NFV2 encoding across a large number of NID scenarios. This breadth of testing helps to support that using AEFR is effective for detecting network attacks.

6.6.1 AEFR with Machine Learning Classifiers

When pairing AEFR with classic machine learning classifiers for NID, we explored all the autoencoder architectures denoted in Table 7 across six NID scenarios. We performed random stratified sampling of 500,000 NetFlow samples from each dataset and used a 70/15/15% split of the data for training, validation, and testing purposes respectively. Min-max normalization was performed on all samples using the training data to determine what values to use for normalization.

In addition to those described in Table 4, we created feature sets that utilize a thresholded version of S. This was performed as an effort to move our results closer to the ideal conditions described in Section 5.5. The thresholding was applied to each sample of S in the following manner. During autoencoder validation, we determine the loss for benign and attack samples separately. We then use those values to determine the thresholds using Equation 55

$$\ell_b + (\ell_a - \ell_b)p \tag{55}$$

where ℓ_b is the benign network flow autoencoder loss, ℓ_a is the attack network flow autoencoder loss, and p indicates a desired percentile between these two values.

We explored thresholds with p set to 0.0, 0.25, 0.5, 0.75, and 1.0. Using these thresholds we performed the threshold operation shown in lines 15-22 of Algorithm 1 on S. We found a large majority of results that showed an improvement over using only X included the thresholded S as part of the features provided to the classifier. We refer to this thresholded version of S as T in our results.

To visualize the impact of thresholding we refer the reader to Figures 29 and 30. Figure 29 shows a set of samples where no thresholding has been applied to S. In this situation, we see that our autoencoder performed better at reconstructing benign network flows than reconstructing attacks. However, there is still data in S_b for benign flows which we would like to eliminate. Shown on the left-hand side of Figure 30 plot (a), using the 50% threshold results in the remaining values of S_b being eliminated while the majority of the attack flows, shown on the right-hand side, have been retained. It is worth noting that the threshold has been found to be sensitive to the amount of separation of loss between the autoencoder reconstruction of benign and attack flows. For instance, on the right-hand side of Figure 30 plot (b), one can see that when using the higher ℓ_a threshold on the same samples depicted in Figure 30 plot (a), we begin to aggressively degrade S_a.

Another example using the 50% threshold is shown in Figure 30 plot (c) to demonstrate that often the thresholding is imperfect, and some benign network flows contain data in S_b. Taken as strictly an anomaly detection framework, this would indicate that those benign flows are likely anomalous. For classification using AEFR, however, a situation like this is not destined to incorrectly identify these benign flows as attacks. Because we are passing the features to a secondary classifier, the feature residuals can be used to attempt the proper classification, whereas the more common method of using a single summary metric of the residuals would

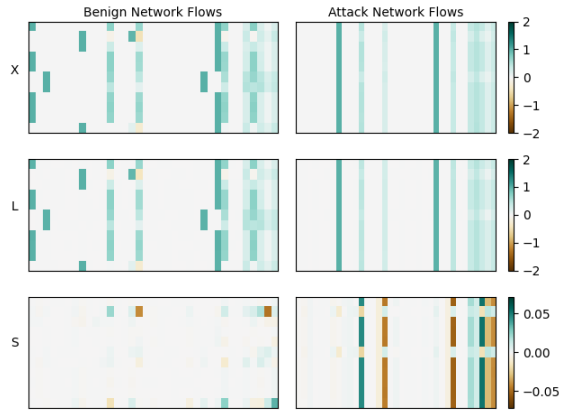


Figure 29: Samples of X, L, and S taken from CTU-13 Scenario 6 network flows using no thresholding on S. We see that the autoencoder performs well but some data still remains in S_b for benign network flows.

not have this opportunity.

After exploring combinations of X, L, and S along with thresholds applied to S, we used initial validation data to determine the most desirable feature sets to use for each scenario. To test this methodology and its robustness, we utilized a completely separate group of data to train these selected models and tested on an independent set of data from each scenario. Final test results are reported in Table 9 where each entry is the mean f1-score from five independent executions of algorithm 1.

Several observations can be made from Table 9 regarding the effectiveness of this technique. First, we observe that in most instances, using our feature combinations and feature residuals with thresholding is able to increase a classifier's f1-score beyond their baseline performance. In many other cases, we at least meet the original baseline values making this technique and its feature sets a viable alternative to simply using the original features. We note that no attempts were made to balance our training data which results in poor performance for some algorithms such as naive Bayes and support vector machine. While performance

Algorithm 1 AEFR training process used when paired with classic machine learning algorithms. The default parameters for these algorithms can be viewed in Appendix A.6. We include all possible features here as input to classifier training. In practice, this could be any combination of the features.

Input: X, p
Output: AE, CLF, Trained autoencoder and classifier.

• Perform data preprocessing on X

$X_b =$ benign samples in X
$X_a =$ attack samples in X

• Initialize autoencoder AE with random weights
• Minimize $mse(X_b - AE(X_b))$ using back propagation

$\ell_b = mse(X_b - AE(X_b))$
$\ell_a = mse(X_a - AE(X_a))$
$t = \ell_b + (\ell_a - \ell_b)p$ ▷ Threshold to be used.
$L = AE(X)$
$S = X - L$

for each sample s in S **do**
    $s\_residual\_mse = \frac{\sum_{i=0}^{N} s_i^2}{N}$
    if $s\_residual\_mse <= t$ **then**
        Set all entries of s to zeros and add row to T
    else
        Add s to T
    end if
end for

• Initialize CLF with default classifier parameters
• Fit CLF using combinations of $X, L, S,$ and T
return AE, CLF

Table 9: Final results pairing AEFR feature sets with classic machine learning algorithms using an independent test set. The default parameters for the algorithms used can be viewed in Appendix A.6. T indicates feature residuals with a threshold applied to them as given in the Threshold column. The top row of each section shows the result using a classifier with just the original features. Values in green indicate an increase in f1-score greater than 0.01. Values in red indicate a decrease in f1-score greater than 0.01. Values in blue indicate no change in f1-score greater than 0.01.

Model Parameters				F1-Score				
Layers	Bottleneck	Features	Threshold	NB	KNN	RF	LR	SVM
CTU13 Scenario 5								
-	-	X	-	0.03	0.76	0.70	0.00	0.00
2	11	L, S	-	0.05 ±0.01	0.75 ±0.01	0.72 ±0.02	0.00 ±0.00	0.00 ±0.00
2	11	X, L, S	-	0.04 ±0.00	0.76 ±0.00	0.73 ±0.02	0.00 ±0.00	0.00 ±0.00
2	11	L, T	75%	0.06 ±0.03	0.74 ±0.01	0.73 ±0.04	0.00 ±0.00	0.00 ±0.00
2	11	X, L, T	75%	0.03 ±0.00	0.75 ±0.01	0.73 ±0.03	0.00 ±0.00	0.01 ±0.02
2	12	L, S	-	0.06 ±0.02	0.76 ±0.01	0.72 ±0.02	0.00 ±0.00	0.00 ±0.00
2	12	X, L, S	-	0.03 ±0.00	0.76 ±0.00	0.71 ±0.02	0.00 ±0.00	0.01 ±0.02
2	12	L, T	75%	0.06 ±0.04	0.75 ±0.01	0.72 ±0.02	0.00 ±0.00	0.00 ±0.00
2	12	X, L, T	75%	0.03 ±0.00	0.75 ±0.01	0.72 ±0.00	0.00 ±0.00	0.01 ±0.02
CTU13 Scenario 6								
-	-	X	-	0.00	0.80	0.86	0.00	0.00
6	12	S	-	0.02 ±0.02	0.87 ±0.00	0.86 ±0.00	0.00 ±0.00	0.00 ±0.00
6	12	T	25%	0.03 ±0.02	0.82 ±0.03	0.81 ±0.03	0.00 ±0.00	0.00 ±0.00
6	12	T	50%	0.03 ±0.03	0.80 ±0.00	0.80 ±0.00	0.00 ±0.00	0.00 ±0.00
6	12	T	75%	0.02 ±0.03	0.48 ±0.44	0.48 ±0.44	0.00 ±0.00	0.00 ±0.00
UNSW-NB15								
-	-	X	-	0.48	0.87	0.88	0.58	0.56
2	9	L, T	25%	0.23 ±0.01	0.87 ±0.00	0.87 ±0.00	0.57 ±0.03	0.60 ±0.02
2	9	X, L, T	25%	0.23 ±0.04	0.87 ±0.00	0.88 ±0.00	0.60 ±0.01	0.64 ±0.01
2	9	X, S, T	25%	0.48 ±0.10	0.87 ±0.00	0.88 ±0.00	0.61 ±0.01	0.64 ±0.02
3	9	L, T	25%	0.27 ±0.10	0.87 ±0.00	0.88 ±0.00	0.54 ±0.04	0.58 ±0.06
3	9	X, L, T	25%	0.27 ±0.10	0.87 ±0.00	0.88 ±0.00	0.61 ±0.00	0.65 ±0.02
3	9	X, S, T	25%	0.50 ±0.12	0.87 ±0.00	0.88 ±0.00	0.61 ±0.00	0.66 ±0.02
4	9	L, T	25%	0.19 ±0.02	0.87 ±0.00	0.87 ±0.00	0.51 ±0.03	0.52 ±0.05
4	9	X, L, T	25%	0.20 ±0.01	0.87 ±0.00	0.87 ±0.00	0.60 ±0.00	0.63 ±0.02
4	9	X, S, T	25%	0.53 ±0.01	0.87 ±0.00	0.88 ±0.00	0.61 ±0.01	0.64 ±0.01
CICIDS2017 Wednesday								
-	-	X	-	0.69	0.98	0.99	0.90	0.90
6	12	X, S	-	0.72 ±0.00	0.98 ±0.00	0.99 ±0.00	0.94 ±0.00	0.94 ±0.01
6	12	X, T	ℓ_b	0.72 ±0.00	0.99 ±0.00	0.99 ±0.00	0.93 ±0.00	0.93 ±0.01
6	12	X, T	25%	0.72 ±0.00	0.99 ±0.00	0.99 ±0.00	0.93 ±0.01	0.94 ±0.01
6	12	X, T	50%	0.72 ±0.00	0.99 ±0.00	0.99 ±0.00	0.93 ±0.00	0.93 ±0.01
6	12	X, T	75%	0.72 ±0.00	0.99 ±0.00	0.99 ±0.00	0.93 ±0.01	0.93 ±0.01
6	12	X, T	ℓ_a	0.72 ±0.00	0.99 ±0.00	0.99 ±0.00	0.93 ±0.01	0.93 ±0.01
CICIDS2017 Thursday								
-	-	X	-	0.02	0.52	0.52	0.00	0.00
6	12	X, S	-	0.03 ±0.00	0.51 ±0.00	0.53 ±0.01	0.00 ±0.00	0.00 ±0.00
6	12	X, T	ℓ_b	0.03 ±0.01	0.52 ±0.00	0.52 ±0.00	0.00 ±0.01	0.01 ±0.01
6	12	X, T	25%	0.03 ±0.00	0.51 ±0.00	0.52 ±0.00	0.00 ±0.00	0.00 ±0.00
6	12	X, T	50%	0.03 ±0.00	0.52 ±0.00	0.52 ±0.00	0.00 ±0.00	0.02 ±0.00
6	12	X, T	75%	0.02 ±0.00	0.52 ±0.00	0.52 ±0.00	0.00 ±0.00	0.03 ±0.03
6	12	X, T	ℓ_a	0.02 ±0.00	0.52 ±0.00	0.52 ±0.00	0.00 ±0.01	0.01 ±0.01
CICIDS2017 Friday								
-	-	X	-	0.73	1.00	1.00	0.98	0.99
6	12	X, S	-	0.79 ±0.03	1.00 ±0.00	1.00 ±0.00	0.99 ±0.00	0.99 ±0.00
6	12	X, T	ℓ_b	0.74 ±0.08	1.00 ±0.00	1.00 ±0.00	0.99 ±0.00	0.99 ±0.00
6	12	X, T	25%	0.76 ±0.05	1.00 ±0.00	1.00 ±0.00	0.99 ±0.00	0.99 ±0.00
6	12	X, T	50%	0.71 ±0.06	1.00 ±0.00	1.00 ±0.00	0.99 ±0.00	0.99 ±0.00
6	12	X, T	75%	0.69 ±0.07	1.00 ±0.00	1.00 ±0.00	0.99 ±0.00	0.99 ±0.00
6	12	X, T	ℓ_a	0.69 ±0.07	1.00 ±0.00	1.00 ±0.00	0.99 ±0.00	0.99 ±0.00

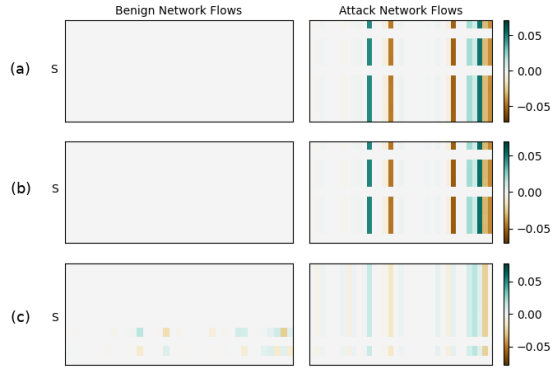


Figure 30: Three examples from CTU-13 Scenario 6 showing the effects of thresholding on S where each row shows S broken out into S_b and S_a. **(a)** A threshold of 50% is applied to the original S from Figure 29. All values for samples of S_b on the left hand side have been eliminated which results in better performance for classification. **(b)** A higher threshold applied to the same samples used in plot (a) begins to degrade S_a shown on the right hand side, which is undesirable. **(c)** An example using samples independent from those used in plots (a) and (b) showing the use of a 50% threshold resulting in some data remaining in S_b on the right hand side. Methods using an aggregate residual metric would suspect the benign samples are anomalies. Our method may still take advantage of passing the feature residuals to a secondary classifier for proper classification.

was still often improved in these circumstances, it was generally not sufficient to make using these algorithms on unbalanced data possible.

It was found that choosing the proper threshold is highly sensitive to the amount of separation between benign and attack residuals. This is observed in Table 9 by looking at the results of using random forest with the CTU13 Scenario 6 data. Here, one can see that as we increase the threshold on S, the performance of the classifier goes from being sufficient to severely degraded. Additionally, we observed that for the UNSW-NB15 dataset, a threshold with p set to 0.25 on S ended up being slightly too high resulting in poor classifier performance using the naive Bayes classifier. Using a slightly lower threshold, the results for this classifier increase significantly from a baseline f1-score of 0.48 to 0.62 across all feature combinations listed in Table 9 for this dataset. Instances where an increase in threshold resulted

in degraded performance caused a mean drop in f1-score of 0.11 using the results in Table 9.

A second cause of classifier performance degradation occurs due to volatility introduced when using L as part of the feature set for classification. This can be observed in Table 9 for the results of the CTU13 Scenario 5 and UNSW-NB15 datasets. Discounting feature combinations that include L yields feature sets that only meet or exceed original classification results. In general, if training data indicates the option of feature sets that include or exclude L, it is recommended to favor the feature sets that exclude L.

6.6.2 AEFR with Neural Networks

Having shown effectiveness on classic machine learning algorithms, we next explored the use of AEFR in combination with several neural network classifiers. The neural network architectures included a feed forward multilayer perceptron (MLP) as well as recurrent architectures such as the RNN and LSTM. In general all of these experiments used stratified random sampling of 500,000 NetFlow records where we used 70/15/15% splits for training, validation, and test respectively. Autoencoders were trained for 500 epochs using the architectures from Table 8.

Our MLP classifier was trained for 100 epochs and was based on the MLP used in Sarhan's work, whose results served as a comparative performance baseline across the datasets explored [90]. The classifier consists of four layers such that each hidden layer contains 10 output nodes with the final layer providing a single output. The ReLU activation function was used for each layer except for the final layer where a sigmoid function was applied. Binary cross entropy was used for the loss function along with the Adam optimizer configured with a learning rate

Table 10: Architecture of the MLP classifiers used. The input layer takes in a multiple of 39 features when we use the NFV2 dataset or a multiple of 35 features for our simple encoding. The remaining layers all have 10 nodes with the final layer providing a single output. We use ReLU for all layers except for the last, where the sigmoid function is applied. The Adam optimizer with a learning rate of 0.001 is used along with the binary cross entropy loss function.

	Input Dimension	**Output Dimension**	**Activation**
Input Layer	Multiple of 39 or 35	10	ReLU
Layer 1	10	10	ReLU
Layer 2	10	10	ReLU
Layer 3	10	1	Sigmoid

of 0.001. Table 10 shows the details of the MLP classifiers.

We first examine our results with an MLP from the perspective of using S as a substitute for X. In Table 11 one can observe the mean f1-scores from ten experiment executions for using X and S as input features to a classifier using the NFV2 family of datasets. We see that the performance of S as a substitute for X yields comparable results with supporting p-values for the NF-UNSW-NB15-V2, NF-BoT-IoT-V2 and NF-CSE-CIC-IDS2018-V2 datasets. For the NF-ToN-IoT-V2 dataset we see that the unadjusted p-value indicates a statistically significant difference in the results with using S being slightly lower than using X. In this case, however, the difference in mean f1-scores between X and S is only 0.003 showing little practical relevance in score differences. Figure 31 shows 100 samples of the NF-CSE-CIC-IDS2018-V2 test data from these results to provide a comparison of X, L, and S. One can observe that the autoencoder does a sufficient job reconstructing benign data and does potentially too well reconstructing attacks. Additionally, there are several large autoencoder feature residuals in the bottom left-hand side of the plot for S_b which have the potential to impact performance when using S for classification. Despite these areas for improvement, we were able to perform

comparable to using the original feature set.

Based on these results one would be able to use S in place of X with little to no change in classification performance. This is an indication that our technique in generating S has maintained the majority of mutual information from X needed to yield comparable results. We find this result fascinating given that we are strictly removing data from X. In addition, in Section 6.9 we show that using S has additional benefits compared to using X in terms of potential data compression.

Another option that the generation of autoencoder feature residuals lends itself to is forming feature sets using S in combination with X and L. We see in Table 11 that combining S with X and L produces results at least as good as using solely X and in several cases the results are improved. For example, we see statistically significant improvements in performance for many of the combinations that include S on the NF-UNSW-NB15-V2 and NF-ToN-IoT-V2 datasets. We do note that these improvements are likely not substantial enough to deem one feature combination to be preferred over the others from a practical perspective.

In a second set of experiments we took the data from the NFV2 datasets and encoded it as described in Table 5 to compare performance across different encodings of the same data. With this encoding, the baseline performance of X is significantly worse on the UNSW-NB15 dataset compared to the original NFV2 dataset, while the other baseline results are comparable. Interestingly, we see in Table 12 that using S or feature combinations including S provides a significant improvement in f1-score with this encoding applied to the UNSW-NB15 data. Similar improvements are seen for the other datasets as well except for the BoT-IoT dataset, where the new feature combinations maintained baseline performance.

Another interesting result that shows the benefit of using S in place of X comes

Table 11: Mean f1-score test results based on ten experiment executions using S in combination with X and L as input to our classifier. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the f1-scores were likely drawn from the same distribution. Values in blue indicate an f1-score and p-value that supports comparable performance compared to X. Values in green indicate an f1-score and p-value that supports we have improved performance compared to X. Values in orange indicate an f1-score and p-value that supports we have degraded performance compared to X. From this table it can be observed that using S alone or in combinations with X and L generally does at least as well as using X alone in terms of classification performance making these combinations of features generally safe to use in place of only using X.

	NF-UNSW-NB15-V2		NF-BoT-IoT-V2		NF-ToN-IoT-V2		NF-CSE-CIC-IDS2018-V2	
	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value
X	0.909		0.989		0.973		0.924	
S	0.904	0.052	0.989	0.418	0.970	0.012	0.924	0.787
XS	0.912	0.052	0.990	0.052	0.975	0.001	0.924	0.168
LS	0.912	0.018	0.990	0.075	0.975	0.011	0.925	0.052
XLS	0.912	0.052	0.990	0.168	0.975	0.003	0.924	0.418

Table 12: Mean f1-score test results based on ten experiment executions using S in combination with X and L as input to our classifier using an alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the f1-scores were likely drawn from the same distribution. Values in blue indicate an f1-score and p-value that supports comparable performance compared to X. Values in green indicate an f1-score and p-value that supports we have improved performance compared to X. It can be seen that using S and feature combinations including S often outperform using X. In other cases these combinations generally perform at least as well as just using X.

	NF-UNSW-NB15-V2		NF-BoT-IoT-V2		NF-ToN-IoT-V2		NF-CSE-CIC-IDS2018-V2	
	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value
X	0.780		0.994		0.950		0.958	
S	0.814	0.012	0.993	0.994	0.953	0.168	0.974	0.000
XS	0.820	0.003	0.995	0.787	0.954	0.012	0.972	0.000
LS	0.825	0.000	0.995	0.418	0.954	0.012	0.973	0.000
XLS	0.819	0.000	0.995	0.787	0.953	0.052	0.969	0.000

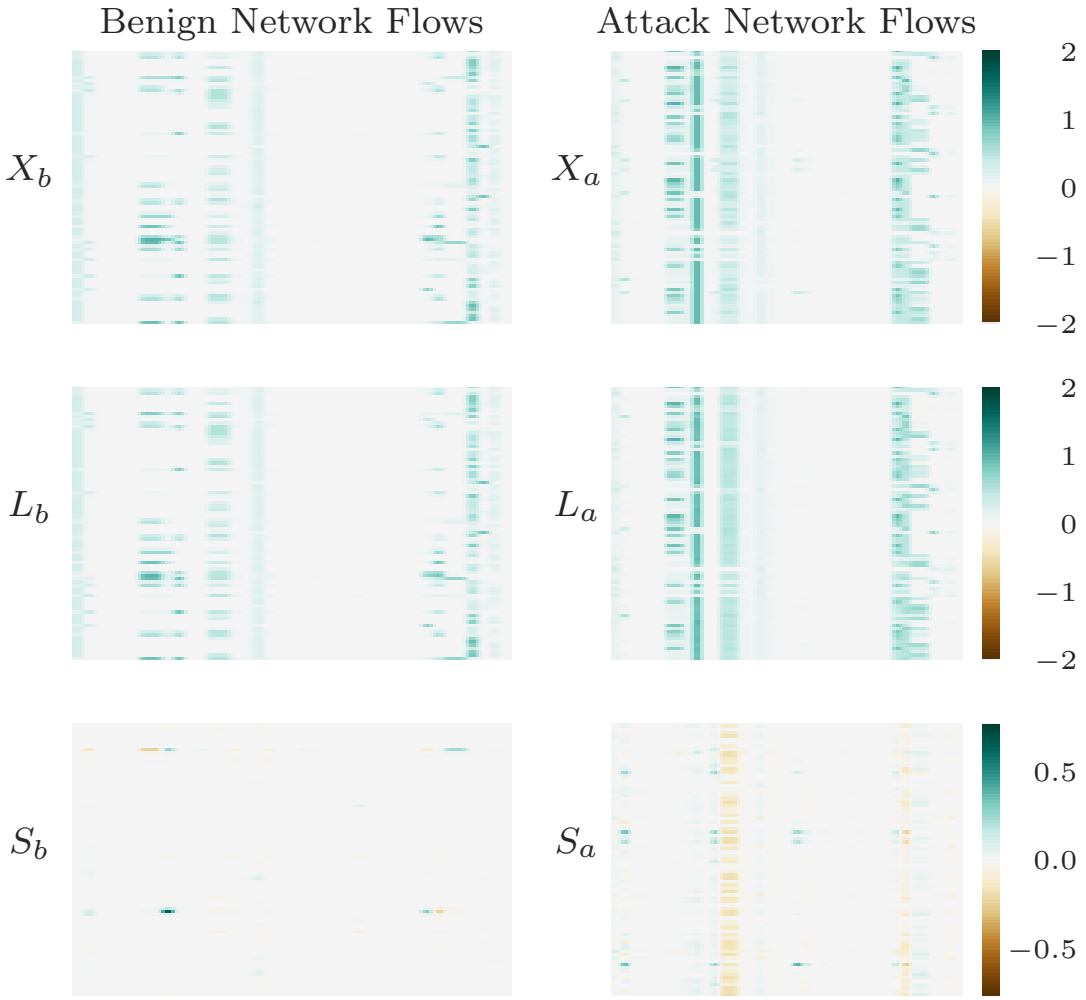


Figure 31: A comparison of X, L, and S using 100 samples with the NF-CSE-CIC-IDS2018-V2 dataset. Using S in place of X provided comparable classification performance. One can observe that the autoencoder has more trouble reconstructing attack network flows on the right-hand side, however, there remain several prominent autoencoder feature residuals visible in S_b in the bottom left-hand side of the plot. These autoencoder feature residuals in S_b have the potential to diminish the performance of using S in place of X for classification, leaving room for improvement such that S has the potential to exceed the performance of X.

when looking at the performance on the CSE-CIC-IDS2018 dataset. With this encoding we have a baseline f1-score of 0.958 which is lower than the original baseline f1-score of 0.97 reported in the work that introduced the NF-CSE-CIC-IDS2018-V2 dataset [91]. Using S instead of X, however, improves our performance to 0.974 allowing this encoding to match that of the NF-CSE-CIC-IDS2018-V2 encoding. This result is compelling as the simple encoding is dominated by one hot encodings along with basic statistics regarding bytes and packets, making it inefficient compared to the NFV2 encoding. One can observe in Figure 32, 100 samples of the simple encoding applied to the NF-CSE-CIC-IDS2018-V2 test data where the autoencoder does extremely well at reconstructing benign network flows compared to attack network flows. The sparsity observed in S_b in the bottom left-hand side of the plot compared to the large autoencoder feature residuals in S_a in the bottom right-hand side of the plot both contribute to the improvement seen over using X for classification. While it remains unclear if using autoencoder feature residuals is specifically effective on simple encodings, this analysis shows that its impact may vary depending on how a dataset is encoded.

Both the RNN and LSTM models used in this work were unidirectional. The model architecture for our recurrent models was found through ablation studies that assessed appropriate sequence lengths, recurrent layers, and hidden state sizes to use. For this work we used a constant sequence length of 25, two recurrent layers with a hidden state size of 24, and a final fully connected layer. Each recurrent layer uses the tanh activation function and a final sigmoid activation is performed on the output of the fully connected layer. We used the default initialization strategy provided by PyTorch[8] for the initialization of hidden and cell states in our recurrent models. Binary cross entropy was used for the loss function

[8]https://pytorch.org/

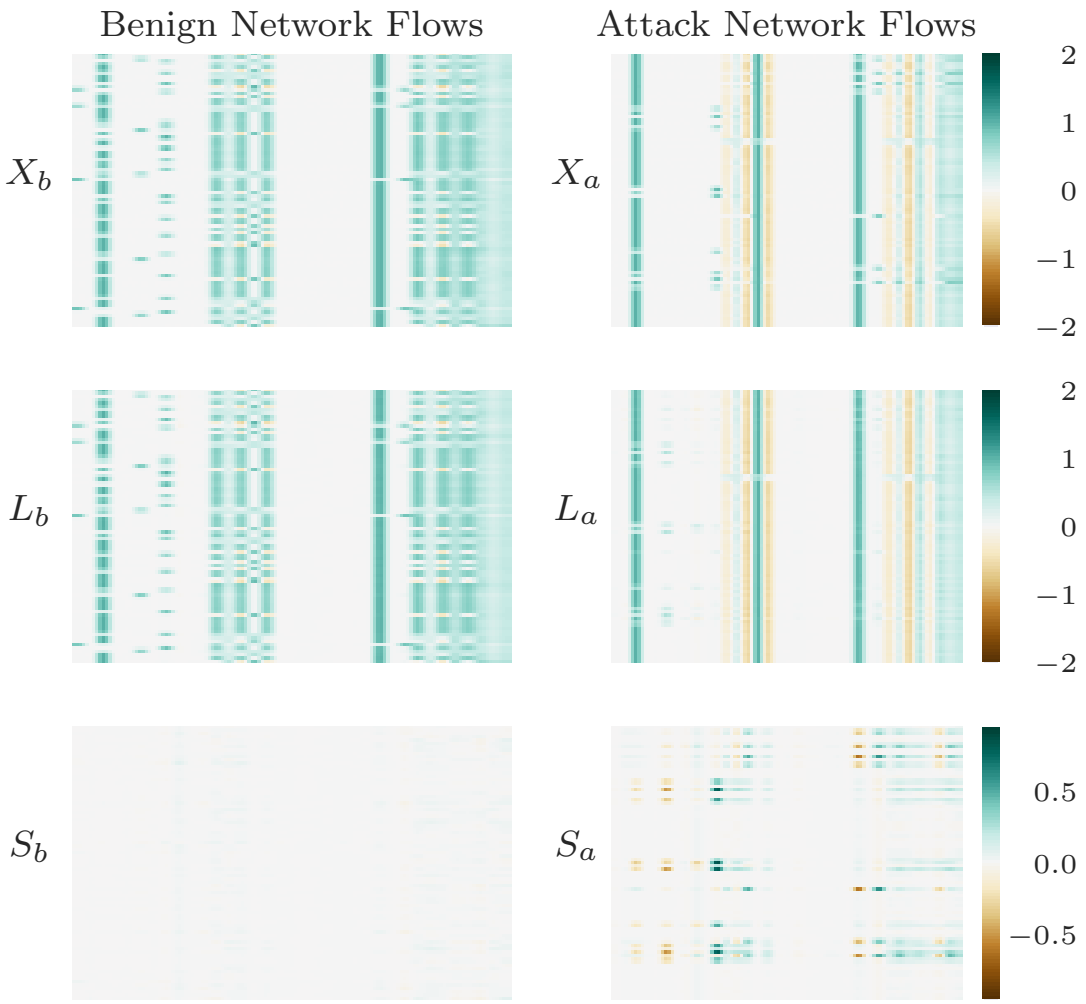


Figure 32: A comparison of X, L, and S using 100 samples with the simple encoding applied to the NF-CSE-CIC-IDS2018-V2 data. Using S with this encoding improved the baseline result compared to using X as input to a classifier, such that it met the original benchmark figures reported on the NF-CSE-CIC-IDS2018-V2 dataset. One can see that S_b on the bottom left side of the plot is sparse with significantly lower autoencoder feature residuals than S_a on the bottom right-hand side.

Table 13: Architecture of the RNN classifiers used. The input layer takes in a multiple of 35 features as these models use the simple encoding of our datasets. The two recurrent layers each have a hidden state size of 24 with the final fully connected layer providing the final output. We use tanh for all recurrent layers and apply a sigmoid activation function to the final fully connected layer. The Adam optimizer with a learning rate of 0.001 is used along with the binary cross entropy loss function. Note that this architecture was used for both the RNN and LSTM models used for classification.

	Input Dimension	**Hidden Dimension**	**Output Dimension**	**Activation**
RNN/LSTM Layer 1	Multiple of 35	24	24	tanh
RNN/LSTM Layer 2	24	24	24	tanh
Fully Connected Layer	24	-	1	Sigmoid

along with the Adam optimizer configured with a learning rate of 0.001. The loss was based on the classifier performance predicting the final NetFlow sample in a given sequence as being attack or benign. The details of the recurrent models are provided in Table 13.

We adjusted our training procedure when using recurrent models such that we utilize 50/25/25% splits (training/validation/test) in order to avoid introducing optimistic results due to any time bias between the training and test data. Additionally, it was found that the recurrent models needed to be trained for 500 epochs. A single sample presented to the recurrent neural networks on which we make a classification consists of 25 consecutive NetFlow samples ordered by flow start time. The features used for each NetFlow sample consisted of our feature sets created using autoencoder feature residuals. For each input sequence we predicted if the final sample in the sequence was attack or benign. This can be seen visualized in Figure 33 where we show a portion of a recurrent model taking in a sequence of NetFlow samples where each sample is represented by X, L, and S.

When using recurrent architectures for our classifier, one can see from Tables 14 and 15 we achieve similar behavior as reported for feed forward networks when

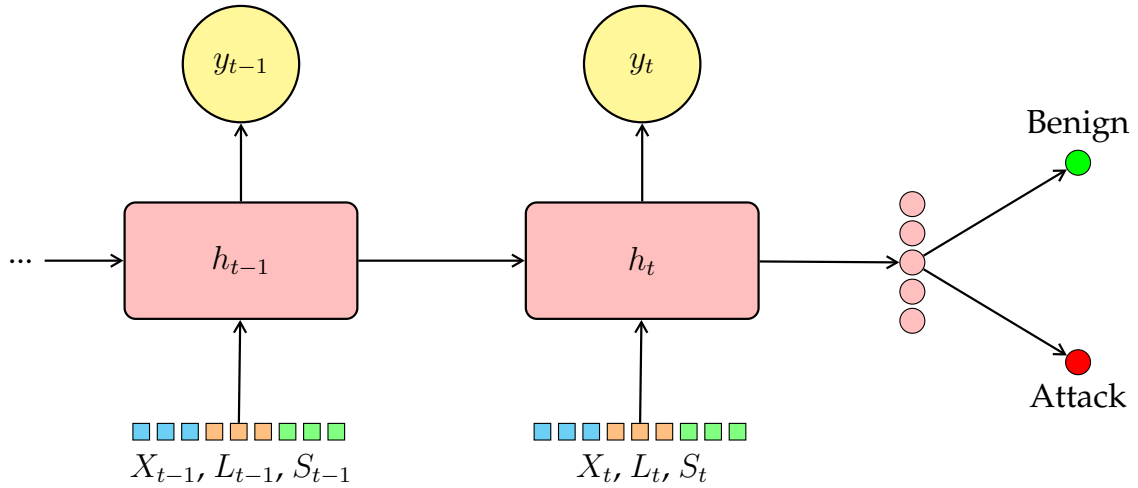


Figure 33: Here we show the general structure for how the features generated using our technique are used with recurrent models. After samples are processed through our autoencoder to create X, L, and S, combinations of these feature sets are used to represent a single network flow. In this example we represent a network flow using all three feature sets; referred to as XLS. A sequence of these samples are provided to the recurrent model. The hidden state of the final unit in the recurrent layers is then provided to a fully connected layer which produces our final prediction of the sequence being an attack or benign sequence. In our experiments we consider a sequence of network flows to be an attack or benign sequence based on the label of the final NetFlow sample in the sequence.

using autoencoder feature residuals as input. In general, using either S on its own, or in combination with other features, we are able to maintain or improve classification performance. While many of these gains are modest we note several practical improvements in f1-score compared to using X, such as using XS as input to an LSTM classifier on the CTU-13 Scenario 13 data. In this case the base f1-score of 0.582 was improved to 0.671. We note a drop in performance on the ToN-IoT dataset when using feature sets that include S without the support of X. However, when S was paired with X in a feature set, our technique achieved a significant performance improvement compared to only using X on that same dataset. Additionally, we see performance improvements using autoencoder feature residuals as input to both RNN and LSTM architectures across all the CTU-13 dataset scenarios demonstrating that this technique may be well-equipped for the detection of botnet attacks.

We note two common conditions encountered when training recurrent models for network intrusion detection. First, it is common to experience some amount of overfitting to training data when working with recurrent models on datasets with a high class imbalance. Additionally, as one moves further out in time for a network intrusion detection scenario, it becomes more likely that the data characteristics change compared to training data [105, 65]. As we were focusing these experiments on getting a general idea for the performance of autoencoder feature residuals on recurrent networks, we put forth little effort to mitigate these two factors. We note this to show that the general model performance improvements reported in Tables 14 and 15 over using X demonstrates that using autoencoder feature residuals provides some benefit in mitigating these factors, while requiring little model tuning compared to other methods.

Table 14: Mean f1-score test results based on ten experiment executions using S in combination with X and L as input to an RNN classifier using the alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the f1-scores were likely drawn from the same distribution. Values in blue indicate an f1-score and p-value that supports comparable performance compared to X. Values in green indicate an f1-score and p-value that supports we have improved performance compared to X. Values in orange indicate an f1-score and p-value that supports we have degraded performance compared to X.

	UNSW-NB15		ToN-IoT		CTU13 Scenario 6		CTU13 Scenario 9		CTU13 Scenario 13	
	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value
X	0.980		0.923		0.826		0.818		0.731	
S	0.979	0.168	0.889	0.012	0.872	0.000	0.828	0.168	0.742	0.418
XS	0.980	0.994	0.934	0.168	0.834	0.168	0.833	0.002	0.801	0.002
LS	0.980	0.787	0.935	0.002	0.834	0.052	0.837	0.000	0.808	0.002
XLS	0.981	0.052	0.927	0.012	0.842	0.052	0.838	0.000	0.792	0.012

Table 15: Mean f1-score test results based on ten experiment executions using S in combination with X and L as input to an LSTM classifier using the alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the f1-scores were likely drawn from the same distribution. Values in blue indicate an f1-score and p-value that supports comparable performance compared to X. Values in green indicate an f1-score and p-value that supports we have improved performance compared to X. Values in orange indicate an f1-score and p-value that supports we have degraded performance compared to X.

	UNSW-NB15		ToN-IoT		CTU13 Scenario 6		CTU13 Scenario 9		CTU13 Scenario 13	
	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value
X	0.976		0.926		0.809		0.778		0.582	
S	0.975	0.052	0.832	0.000	0.816	0.168	0.796	0.002	0.639	0.012
XS	0.976	0.418	0.952	0.012	0.828	0.168	0.797	0.000	0.671	0.002
LS	0.976	0.787	0.919	0.012	0.822	0.418	0.797	0.000	0.664	0.002
XLS	0.977	0.052	0.951	0.002	0.835	0.012	0.798	0.000	0.654	0.012

While the focus of these experiments is on using S, we do recognize that one could also use L or XL as feature combinations in place of X. We found that in general XL performed comparable to X, however, L generally performed worse than X when using an MLP, RNN, or LSTM classifier. We believe this is in line with our findings for classic machine learning algorithms; in that the usage of L, or combinations with L, tends to hold some volatility in results dependent on the differences of benign training samples compared to the test data used.

6.6.3 AEFR for One-class Classification

Having already explored a number of supervised algorithms and neural networks, we sought to explore methods that could make the entire path of identifying network attacks possible by only training with benign network flow samples. To do this, we alter our training such that the downstream classifier is only provided benign network flow samples similar to how we train our autoencoder. This is of significance for NID as there is mainly an abundance of benign network data available, however, quality attack data is generally scarce. In our one-class classification experiments we continue using the same feature encodings along with several Internet of Things (IoT) datasets to test the usage of AEFR with one-class algorithms. These algorithms include isolation forest, one-class support vector machine, and local outlier factor. Due to the importance of one-class classification in this area we first examine baseline results with no algorithm tuning. We then performed ablations across the downstream classifier parameters for further analysis of the effectiveness of pairing AEFR with one-class classifiers. For all experiments we take a random stratified draw based on the class labels of no more than 500,000 samples and use splits of 70/15/15% corresponding to training, validation, and

test data respectively. The autoencoder training process remains unchanged from our previous experiments described in this chapter, using only benign data from the training split. Our final results continue to utilize the hold out test split to obtain our final reported performance measures.

We first look at the results of each feature combination in comparison to just using X as input to each classifier. In these initial experiments no tuning was done to the classifiers and so the default parameters selected by the sklearn[9] libraries were used. These parameters and their default values can be found in Appendix A.6. Table 16 shows our results across seven IoT scenarios using our technique as input to three different algorithms which we trained as one-class classifiers. From the table one can see that using our feature sets generated with autoencoder feature residuals generally outperforms X or at a minimum performs at least as well. We note several cases where using S provides a significant performance improvement, such as when using IF on the BoT-IoT dataset, where using X received an f1-score of 0.393 and S was able to achieve an f1-score of 0.770. Similar significant improvements were realized when assessing OCSVM on the BoT-IoT dataset as well as when applying LOF on the simple encoding of the ToN-IoT dataset. In cases such as these, it may be possible to use S in place of X as an alternative to extensive hyperparameter tuning of X.

Denoted in orange in Table 16 are baseline results that did not perform as well as X. For many of these results the differences are negligible in practical terms, such as the application of OCSVM to Scenario 13 of the IoT-23 dataset, where S achieved an f1-score of 0.787 compared to 0.789 for X indicating only a difference of 0.002. We note the results when applying IF and OCSVM to the ToN-IoT dataset where both X and S perform poorly. These results imply that when X requires

[9]https://scikit-learn.org/stable/index.html

Table 16: Baseline performance measures taken using default parameters for each one-class classifier. We report mean f1-score test results based on ten experiment executions using S in combination with X and L as input to our classifier. P-values using the two sample Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the f1-scores were likely drawn from the same distribution. Values in blue indicate an f1-score and p-value that supports comparable performance compared to X. Values in green indicate an f1-score and p-value that supports we have improved performance compared to X. Values in orange indicate an f1-score and p-value that supports we have degraded performance compared to X. We see that using S or S along with X and L often provides significant performance increases compared to only using X and at a minimum does not harm classification performance.

	ToN-IoT		BoT-IoT		ToN-IoT (Simple)		IoT-23 Scenario 1		IoT-23 Scenario 13		IoT-23 Scenario 19		IoT-23 Scenario 20	
	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value	F1-Score	p-value
Isolation Forest														
X	0.303	-	0.393	-	0.946	-	0.939	-	0.887	-	0.598	-	0.679	-
S	0.063	0.000	0.770	0.000	0.751	0.000	0.950	0.052	0.879	0.052	0.605	0.012	0.631	0.052
XS	0.120	0.343	0.527	0.502	0.926	0.168	0.947	0.168	0.877	0.052	0.624	0.168	0.646	0.052
LS	0.111	0.169	0.622	0.001	0.877	0.002	0.928	0.168	0.898	0.012	0.709	0.994	0.675	0.418
XLS	0.111	0.160	0.510	0.905	0.933	0.168	0.930	0.012	0.868	0.012	0.647	0.787	0.669	0.418
One-class Support Vector Machine														
X	0.605	-	0.793	-	0.938	-	0.886	-	0.789	-	0.628	-	0.662	-
S	0.485	0.000	0.934	0.000	0.987	0.000	0.881	0.002	0.787	0.000	0.689	0.012	0.847	0.000
XS	0.597	0.001	0.893	0.567	0.938	0.000	0.887	0.002	0.788	0.000	0.754	0.000	0.777	0.000
LS	0.617	0.000	0.889	0.038	0.938	0.000	0.887	0.000	0.790	0.000	0.869	0.000	0.709	0.168
XLS	0.607	0.399	0.866	0.333	0.938	0.000	0.887	0.000	0.790	0.000	0.847	0.000	0.705	0.168
Local Outlier Factor														
X	0.365	-	0.861	-	0.758	-	0.951	-	0.918	-	0.893	-	0.737	-
S	0.406	0.000	0.888	0.000	0.936	0.000	0.950	1.000	0.918	1.000	0.898	0.787	0.705	0.168
XS	0.368	0.168	0.880	0.000	0.770	0.002	0.950	0.994	0.918	1.000	0.895	0.994	0.737	0.052
LS	0.361	0.052	0.879	0.000	0.763	0.002	0.950	0.994	0.918	1.000	0.897	0.168	0.738	0.052
XLS	0.361	0.052	0.883	0.000	0.763	0.052	0.951	1.000	0.918	1.000	0.900	0.418	0.738	0.052

additional hyperparameter tuning, the same may apply to S.

Based on our initial baseline results, we explored tuning the algorithms in efforts to compare using each algorithm optimized for X as well as for our generated feature combinations. We performed ablations across most parameters of the algorithms but found the contamination, nearest neighbors, and kernel parameters, as summarized in Figure 34, to be most significant in terms of increasing performance for these algorithms. In addition, we performed grid searches across combinations of parameters but generally found no significant increases in performance com-

pared to that depicted in Figure 34. There were several key takeaways learned from performing these studies in terms of the dynamics of X and S. First, we found that in general, after performance tuning S and X individually, S is likely to outperform X. In nearly all other instances, S or a feature combination including S at least meets the performance of X.

Along with these results, we found that the algorithm parameters that work well with X are not always the same as the parameters that result in a high performing S. When applying various levels of contamination across all datasets for the LOF algorithm, both X and S track together with performance increasing or decreasing depending on the level of contamination indicated. For an algorithm such as OCSVM, however, we observe that S is much more sensitive to the selected kernel compared to X, and it is often not the same kernel that works best for both of them. This can be seen by the results of the BoT-IoT dataset, where S performed best with the rbf kernel, while X showed its best result using a polynomial kernel.

In general, these ablation studies show that with little risk, one can simply tune S or a feature combination using S in place of X and likely increase their classification performance for IoT network intrusion detection.

We now provide two comparisons to the best results achieved after performance tuning X across all datasets. As shown in Table 17, we have a direct comparison of using S with the same algorithm that yielded the best performance with X as its input. We then show our best performance across all the feature combinations and algorithms assessed during our evaluation. From the direct comparison of X and S one can see that S generally outperforms X in many cases and meets the performance of X in other cases. This makes it a suitable substitute for X provided that we independently perform a hyperparameter search for S on the same

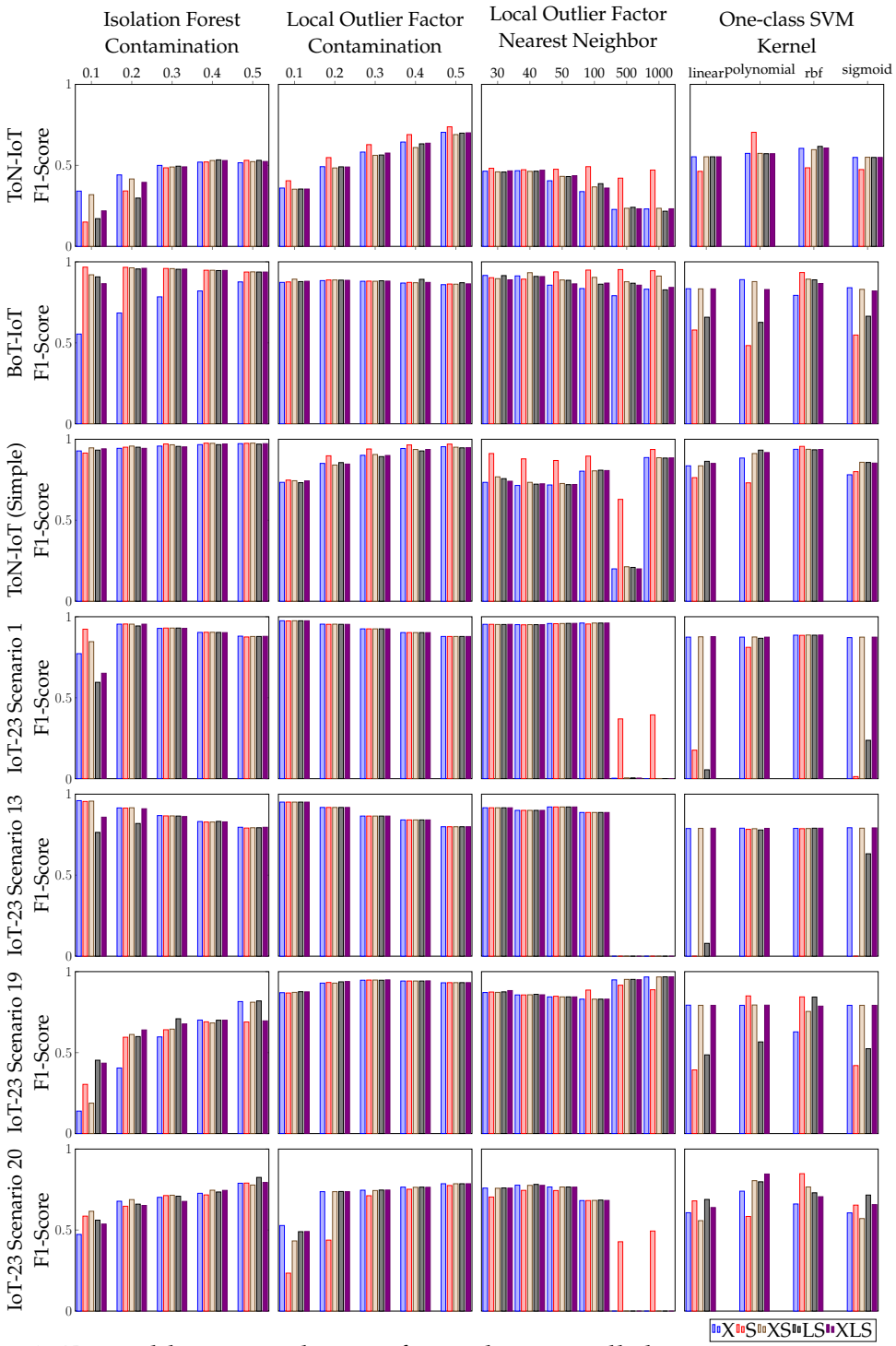


Figure 34: Four ablation studies performed across all datasets to compare using feature combinations that include S to just using X when the one-class classification algorithms have been optimized for performance. In general, these ablation studies show that our novel feature combinations generally outperform X or meet the same performance of X even when the classifiers are optimized beyond their baseline settings.

Table 17: A comparison of the best performance found using X compared to S by itself, as well as all of the assessed feature combinations constructed by our technique. We note the highest mean f1-score in bold. It can be seen that in general the feature combinations constructed with our technique outperforms X. Additionally, we can see that S on its own often provides improved performance over X.

	Best X Algorithm	X Parameters	S Parameters	X F1-Score	S F1-Score	Our Best Algorithm	Our Parameters	Our Best F1-Score
ToN-IoT	Local Outlier Factor	contamination = 0.5	contamination = 0.5	0.704	**0.738**	Local Outlier Factor	S; contamination = 0.5	**0.738**
BoT-IoT	Local Outlier Factor	neighbors = 30	neighbors = 500	0.916	0.952	Isolation Forest	S; contamination = 0.1	**0.967**
ToN-IoT (Simple)	Isolation Forest	contamination = 0.5	contamination = 0.4	0.973	0.976	One-class SVM	S; kernel = rbf	**0.987**
IoT-23 Scenario 1	Local Outlier Factor	contamination = 0.1	contamination = 0.1	**0.975**	0.974	Local Outlier Factor	XS; contamination = 0.1	**0.975**
IoT-23 Scenario 13	Isolation Forest	contamination = 0.1	contamination = 0.1	**0.960**	0.955	Isolation Forest	XS; contamination = 0.1	0.957
IoT-23 Scenario 19	Local Outlier Factor	neighbors = 1000	contamination = 0.3	0.967	0.948	Local Outlier Factor	LS; neighbors = 1000	**0.968**
IoT-23 Scenario 20	Isolation Forest	contamination = 0.5	contamination = 0.5	0.789	0.790	One-class SVM	S; kernel = rbf	**0.848**

algorithm. Additionally, we see that opening up our consideration of feature combinations and algorithms, using our technique was able to meet or exceed the best performing X. The one exception occurred with the IoT-23 Scenario 19 dataset where X achieved an f1-score of 0.960 while we achieved 0.957, which we do not consider significant when using this technique in practical IoT settings.

Taking an intuitive look at our results, we find it fascinating that using S and its accompanying feature combinations are able to meet and often surpass the performance of X. In our process, S is derived from X, meaning that no new data is generated and actually, data is removed. This implies that the generation of S is removing non-essential portions of X and presenting features to downstream classifiers in a more efficient representation.

6.7 AEFR Precision and Recall Summary

As a summary metric we report mean f1-scores for our experiments, as it serves as a useful comparative metric. Along with this we also collect precision and recall metrics which are used to calculate the f1-score. These collected results are provided in Appendix A.7.2 and we provide a summary of the insights they provide

here.

There are several takeaways that shed light into how AEFR increases overall classification performance. First, we note that in general for neural networks, our feature combinations tend to increase precision while maintaining respectable recall measures. When looking at using AEFR with one-class classifiers, we perform well in both precision and recall, while we note inconsistent results with our feature combinations when the baseline metrics for X would be considered undesirable. Interestingly, our ablation studies with one-class classifiers found that using AEFR nearly always outperformed X in terms of recall, while, maintaining precision metrics compared to X.

We note that these findings align with the previously reported f1-scores as precision and recall are used to calculate that metric. Additionally, we note the improvements in precision performance align with our results to be reported in Section 6.8, in that our increases in precision align with a reduction of false alarm rate.

6.8 AEFR Impact on False Alarm Rate

One of the key metrics to consider when performing NID is the false alarm rate (FAR) as defined in Appendix A.2.5. Often called the false positive rate in other settings, this metric has significant implications to the practical application of NID. When a potential network intrusion is detected by a NIDS, this generally triggers an alert to network operators for investigation. If a large number of these alerts are benign and not actually attacks, it causes additional work for the network operators. Additionally, when this is severe, operators may experience false alarm fatigue and ignore many alerts that could be potential attacks. For this reason, we pay special attention to to minimizing this metric in the area of NID.

Table 18: Mean false alarm rate (FAR) based on ten experiment executions using S in combination with X and L as input to our MLP classifier whose classifications results are reported in Table 11. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the false alarm rate were likely drawn from the same distribution. Values in blue indicate a false alarm rate and p-value that supports comparable performance compared to X. Values in green indicate a false alarm rate and p-value that supports we have decreased the false alarm rate compared to X. Values in orange indicate a false alarm rate and p-value that supports we have increased the false alarm rate compared to X.

	NF-UNSW-NB15-V2		**NF-BoT-IoT-V2**		**NF-ToN-IoT-V2**		**NF-CSE-CIC-IDS2018-V2**	
	FAR	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**
X	0.024		0.020		0.114		0.000	
S	0.025	0.873	0.028	0.357	0.126	0.357	0.000	1.000
XS	0.023	0.873	0.021	0.873	0.099	0.873	0.000	1.000
LS	0.025	0.873	0.020	1.000	0.109	0.873	0.000	1.000
XLS	0.023	0.873	0.020	1.000	0.106	0.357	0.000	0.873

For the MLP classification experiments reported in Section 6.6.2 we show the associated FAR results in Tables 18 and 19. In general, the baseline FAR when using X is at an acceptable level across the datasets tested, which is somewhat expected due to these being supervised algorithms. We can also see from these results that the use of AEFR has either maintained this baseline FAR or improved it. In particular we can see in Table 19 we improve the FAR by 1% when using S for the NF-CSE-CIC-IDS2018-V2 dataset with the simple encoding.

We examine the FAR metrics collected along with our experiments that utilized RNN and LSTM classifiers in Tables 20 and 21 respectively. Similar to the results obtained with MLPs, we have a generally low FAR for our baseline results using X. We note that in this context our feature combinations generated using AEFR generally lowered the FAR though often only in modest amounts. It is interesting to see that on its own, using S on the ToN-IoT dataset increased the FAR with both the RNN and LSTM classifiers. When used in combination with other supporting

Table 19: Mean false alarm rate (FAR) based on ten experiment executions using S in combination with X and L as input to our MLP classifier using an alternative simple encoding whose classification results are reported in Table 12. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the false alarm rate were likely drawn from the same distribution. Values in blue indicate a false alarm rate and p-value that supports comparable performance compared to X. Values in green indicate a false alarm rate and p-value that supports we have decreased the false alarm rate compared to X. Values in orange indicate a false alarm rate and p-value that supports we have increased the false alarm rate compared to X.

	NF-UNSW-NB15-V2		**NF-BoT-IoT-V2**		**NF-ToN-IoT-V2**		**NF-CSE-CIC-IDS2018-V2**	
	FAR	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**
X	0.145		0.075		0.214		0.017	
S	0.122	0.357	0.064	0.357	0.199	0.873	0.007	0.008
XS	0.125	0.357	0.059	0.079	0.204	0.873	0.010	0.008
LS	0.140	1.000	0.059	0.079	0.210	0.873	0.009	0.008
XLS	0.117	0.357	0.066	0.873	0.201	0.873	0.010	0.008

features, however, our features provide a significant reduction in FAR with a reduction of 8.2% when using XLS for the RNN classifier and a reduction of 9.4% when using LS for the LSTM classifier.

Examining the FAR metrics collected along with our baseline experiments using one-class classifiers we see that the baseline FAR using X reported in Table 22 is often considerably higher compared to the previously discussed supervised classifiers. This is in general expected when working with algorithms that only use partial labels. From these results we can see that using our feature combinations often provide significant improvements or maintain the baseline FAR. In the instances where using our feature sets results in an increase in FAR, the difference from the baseline using X is often within 1%. In contrast, when we decrease the FAR, it is often by more than 1% with the greatest improvement occurring on the ToN-IoT (Simple Encoding) dataset where we decreased the FAR by 14.9% using

Table 20: Mean false alarm rate (FAR) test results based on ten experiment executions using S in combination with X and L as input to an RNN classifier using the alternative simple encoding whose classification results are reported in Table 14. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the false alarm rates were likely drawn from the same distribution. Values in blue indicate a false alarm rate and p-value that supports comparable performance compared to X. Values in green indicate a false alarm rate and p-value that supports we have decreased the false alarm rate compared to X. Values in orange indicate a false alarm rate and p-value that supports we have increased the false alarm rate compared to X.

	UNSW-NB15		**ToN-IoT**		**CTU13 Scenario 6**		**CTU13 Scenario 9**		**CTU13 Scenario 13**	
	FAR	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**
X	0.010		0.151		0.001		0.047		0.009	
S	0.009	0.000	0.161	0.000	0.001	0.000	0.040	0.000	0.005	0.000
XS	0.009	0.000	0.099	0.000	0.001	0.000	0.043	0.000	0.006	0.000
LS	0.009	0.000	0.085	0.000	0.001	0.000	0.039	0.000	0.005	0.000
XLS	0.009	0.000	0.069	0.000	0.001	0.000	0.039	0.000	0.006	0.000

Table 21: Mean false alarm rate test results based on ten experiment executions using S in combination with X and L as input to an LSTM classifier using the alternative simple encoding whose classification results are reported in Table 15. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the false alarm rates were likely drawn from the same distribution. Values in blue indicate a false alarm rate and p-value that supports comparable performance compared to X. Values in green indicate a false alarm rate and p-value that supports we have decreased the false alarm rate compared to X. Values in orange indicate a false alarm rate and p-value that supports we have increased the false alarm rate compared to X.

	UNSW-NB15		**ToN-IoT**		**CTU13 Scenario 6**		**CTU13 Scenario 9**		**CTU13 Scenario 13**	
	FAR	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**	**FAR**	**p-value**
X	0.012		0.153		0.001		0.058		0.013	
S	0.012	0.000	0.201	0.000	0.001	0.000	0.051	0.000	0.010	0.000
XS	0.012	0.000	0.081	0.000	0.001	0.000	0.053	0.000	0.009	0.000
LS	0.012	0.000	0.059	0.000	0.001	0.000	0.052	0.000	0.009	0.000
XLS	0.012	0.000	0.094	0.000	0.001	0.000	0.050	0.000	0.010	0.000

S as input to the Isolation Forest classifer.

When comparing the FAR metrics which correlate to the best performing one-class classifiers using X and our feature sets, we see that our method generally meets or exceeds the FAR obtained when using X as reported in Table 23. In the two instances where X has a more favorable FAR compared to feature combinations using AEFR, we increase the FAR by 1.8% and 0.4% when considering our best performing algorithms, maintaining respectable performance.

Overall, our analysis of the FAR when using AEFR compared to using X shows that similar to classification performance, our method generally improves FAR metrics or attains the same performance. In instances where we do increase the FAR, it was found to be by small amounts. In contrast, in instances where using AEFR reduced the FAR, our technique provided comparatively larger improvements. These results show that when applying AEFR to NID, we need not worry about causing a significant increase in FAR and maintain the potential to reduce the FAR when compared to using X.

6.9 Compression Properties of AEFR

As noted throughout our results, there are times when S performs only as good as X, which leads to a question of determining if there are any benefits for using S if we knew those cases a priori. One such benefit is the potential for compressing collected data and reducing the load on the collection resources of a network. We outline a simple set of assumptions to allow for a brief analysis of the compression of S. If inserted into the collection pipeline of a NIDS, one could reduce the amount of data required to be collected by forgoing collection on the features of X and only collect S. For this analysis, we consider only benign network samples as that

Table 22: Baseline performance measures taken using default parameters for each one-class classifier whose classification results are reported in Table 16. We report mean false alarm rate (FAR) test results based on ten experiment executions using S in combination with X and L as input to our classifier. P-values using the two sample Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the false alarm rates were likely drawn from the same distribution. Values in blue indicate a false alarm rate and p-value that supports comparable performance compared to X. Values in green indicate a false alarm rate and p-value that supports we have decreased the false alarm rate compared to X. Values in orange indicate a false alarm rate and p-value that supports we have increased the false alarm rate compared to X.

	ToN-IoT		BoT-IoT		ToN-IoT (Simple)		IoT-23 Scenario 1		IoT-23 Scenario 13		IoT-23 Scenario 19		IoT-23 Scenario 20	
	FAR	p-value	FAR	p-value	FAR	p-value	FAR	p-value	FAR	p-value	FAR	p-value	FAR	p-value
Isolation Forest														
X	0.068	-	0.056	-	0.211	-	0.231	-	0.237	-	0.279	-	0.227	-
S	0.035	0.000	0.031	0.000	0.062	0.000	0.183	0.052	0.257	0.052	0.263	0.012	0.192	0.168
XS	0.038	0.000	0.034	0.000	0.084	0.000	0.199	0.168	0.261	0.052	0.260	0.418	0.190	0.012
LS	0.064	0.106	0.048	0.012	0.076	0.000	0.247	0.787	0.215	0.012	0.285	0.168	0.241	0.168
XLS	0.054	0.012	0.044	0.007	0.096	0.000	0.253	0.052	0.285	0.012	0.282	0.787	0.209	0.418
One-class Support Vector Machine														
X	0.500	-	0.496	-	0.513	-	0.457	-	0.498	-	0.484	-	0.530	-
S	0.502	0.000	0.497	0.012	0.512	0.012	0.479	0.002	0.505	0.000	0.496	0.000	0.520	0.052
XS	0.498	0.000	0.494	0.012	0.513	0.418	0.456	0.002	0.502	0.000	0.485	0.168	0.532	0.418
LS	0.499	0.001	0.497	0.052	0.498	0.012	0.453	0.000	0.495	0.000	0.488	0.012	0.527	0.168
XLS	0.499	0.000	0.495	0.168	0.510	0.000	0.453	0.000	0.495	0.000	0.487	0.002	0.538	0.168
Local Outlier Factor														
X	0.105	-	0.073	-	0.133	-	0.184	-	0.167	-	0.113	-	0.270	-
S	0.111	0.002	0.113	0.000	0.164	0.000	0.185	1.000	0.167	1.000	0.116	0.418	0.272	0.418
XS	0.105	0.168	0.078	0.000	0.135	0.002	0.186	0.994	0.167	1.000	0.114	0.787	0.273	0.052
LS	0.107	0.012	0.077	0.000	0.130	0.000	0.187	0.994	0.167	1.000	0.113	1.000	0.269	0.418
XLS	0.105	0.052	0.076	0.000	0.131	0.002	0.184	1.000	0.167	1.000	0.114	0.787	0.268	0.052

Table 23: For each of the best performing one-class classifiers from Table 17 we report the mean false alarm rate (FAR). We note the lowest mean FAR in bold.

	Best X Algorithm	X Parameters	S Parameters	X FAR	S FAR	Our Best Algorithm	Our Parameters	Our Best Algorithm's FAR
ToN-IoT	Local Outlier Factor	contamination = 0.5	contamination = 0.5	**0.539**	**0.539**	Local Outlier Factor	S; contamination = 0.5	**0.539**
BoT-IoT	Local Outlier Factor	neighbors = 30	neighbors = 500	**0.077**	0.231	Isolation Forest	S; contamination = 0.1	0.095
ToN-IoT (Simple)	Isolation Forest	contamination = 0.5	contamination = 0.4	0.508	**0.410**	One-class SVM	S; kernel = rbf	0.508
IoT-23 Scenario 1	Local Outlier Factor	contamination = 0.1	contamination = 0.1	**0.092**	0.093	Local Outlier Factor	XS; contamination = 0.1	**0.092**
IoT-23 Scenario 13	Isolation Forest	contamination = 0.1	contamination = 0.1	**0.079**	0.089	Isolation Forest	XS; contamination = 0.1	0.083
IoT-23 Scenario 19	Local Outlier Factor	neighbors = 1000	contamination = 0.3	0.259	0.320	Local Outlier Factor	LS; neighbors = 1000	**0.253**
IoT-23 Scenario 20	Isolation Forest	contamination = 0.5	contamination = 0.5	0.541	0.537	One-class SVM	S; kernel = rbf	**0.524**

would generally dominate the data collected on a network unless it is actively under attack. Due to the fact that S is derived from X we will assume that the features of S and X require the same amount of base storage. We then perform our comparison by eliminating any values that are zero, assuming they contain no interesting data, as would be the case if our autoencoder perfectly reconstructed a feature within a sample. While there would need to be some form of meta-data to describe what data is present in the compressed format, we assume this is equal for both X and S, allowing us to simply analyze a sampling of our data by eliminating all values that are zero.

Taking 500 samples of X and S we report the mean compression ratios in Table 24. In doing this it was found that due to our current autoencoder structure, no values in S were truly zero while many were minuscule and could be considered zero. To account for this we empirically determined that any values greater than -0.001 and less than 0.01 could be considered zero without impact to classification performance, and were treated as such for this analysis.

Looking at the compression ratios of the thresholded S, we see that it generally outperformed the compression possible on X based on our assumptions. A higher compression rate was found for X on the NF-BoT-IoT-V2 dataset compared to that of the thresholded S which is likely due to suboptimal thresholding options for that particular dataset. We leave exploring a more definitive way of thresholding S and regularizing the autoencoder to induce values of zero in the feature residuals to future work. While this was a simplistic view of a compression scenario, it shows that S has potential benefits over X beyond classification performance gains that could be explored further in future research.

Table 24: Mean compression ratios based on 500 samples of benign data from each dataset. For this analysis we removed any values that were zero in the samples. We see that S does not naturally have exact zeros that can be removed so small positive and negative thresholds were used to determine what to drop from S. We see that overall, a thresholded S has a higher compression ratio compared to compressing X in the same manner. In parentheses we show the mean f1-scores from these experiments which were comparable to those reported in Table 11 for S.

	NF-UNSW-NB15-V2	**NF-BoT-IoT-V2**	**NF-ToN-IoT-V2**	**NF-CSE-CIC-IDS2018-V2**
X (No Zeros)	1.857	2.254	2.313	2.326
S (No Zeros)	0.000	0.000	0.000	0.000
Thresholded S (No Zeros)	3.168 (0.903)	2.074 (0.989)	3.331 (0.964)	5.075 (0.924)

6.10 AEFR Efficiency Discussion

Having reviewed the detailed results across a wide variety of datasets, downstream classifiers, and data encodings, we wanted to summarize some of the recurring themes that have been found regarding the usage of AEFR in our research. First, and perhaps most surprising, is that these results provide confidence that one can use S in place of X for NID with essentially no risk to performance. Contrary to this, however, usage of L results in volatile performance and often performs worse than X. Finally, we want to reinforce the notion that among all of our feature combinations, no new data is generated as we derive all of it from X. This implies that using AEFR tends to remove inefficiencies in X that make the data more accessible to downstream models.

7 Improving the Network Intrusion Detection Dataset Workflow

Applying deep learning techniques to perform network intrusion detection has expanded significantly in recent years. Throughout our research we relied on the availability of sufficient NID datasets. Despite recent improvements to these datasets, it remains difficult to effectively compare methodologies across a wide variety of datasets due to the unique features generated as part of the delivered datasets. In addition, it is often difficult to generate new features using a dataset due to the lack of source data or inadequate ground truth labeling information for a given dataset. These limitations have been identified by a number of researchers in recent literature [24, 32, 90, 91, 88, 91, 106]. In this chapter we seek to reduce the impact of limitations that occur as a result of the handoff of NID datasets from a dataset developer to downstream NID researchers. We propose a set of guidelines to help dataset developers overcome handoff limitations and extend the positive impact these datasets can have on downstream researchers. Our focus on the handoff of NID datasets between researchers has not been well explored in current NIDS dataset research. While many of the guidelines are generic in nature, we provide details on how to specifically implement them for NID datasets. In addition to the guidelines, we developed an open source containerized environment and framework to support implementation of the guidelines[10].

Figure 35 shows the dataset development process adapted from descriptions in recent research to show where this research logically fits [90, 52]. As can be seen highlighted in the figure, we focus on improvements for NIDS dataset feature and

[10]https://github.com/WickedElm/niddff

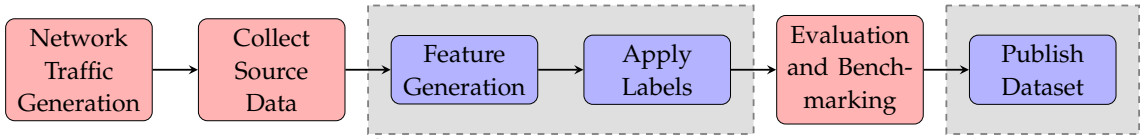


Figure 35: An overview of the NID dataset development process. The areas that our research seeks to improve are depicted in blue with a gray background.

label generation which leads to additional improvements for the final delivery of the dataset. The provided set of guidelines can be used such that the end delivery of a NIDS dataset includes the original source network data as well as concrete scripts for generating each feature and label. With both of these items in hand, researchers will be able to reliably recreate a dataset from source data, ensure the same features are available across multiple datasets, and perform additional feature engineering.

7.1 NID Dataset Development Tools

One of the earliest tools concerned with NID datasets was FLAME [16]. The main goal of the FLAME tool was to take existing NetFlow data and augment it by injecting new anomalies into the existing flows. In doing this, it would allow researchers to capture live network traffic and then augment it later with anomalies, resulting in a dataset usable for developing NID methodologies.

With a goal similar to FLAME, the ID2T tool also focused on augmenting network data with attacks [26, 27]. The ID2T tool, however, approached this from the packet level, ingesting PCAP files as opposed to NetFlow data. In addition, the ID2T tool is capable of providing reports regarding the attacks injected such that they can be used to facilitate data labeling. These qualities allowed ID2T to be capable of injecting a larger variety of network attacks making it useful for the generation of new NID datasets that combine live network traffic with synthetic

attacks.

The INSecS-DCS tool [80] is another NID dataset creation tool which has an expanded scope compared to both FLAME and ID2T. Rather than focus on injecting attacks, INSecS-DCS focuses on processing packet data, live or from a PCAP file, such that one can customize the features to include in a final processed dataset. These features can be captured as packet level statistics or based on time windows.

Another related work introduces NDCT [2], which provides a toolkit for the collection and annotation of cybersecurity datasets. NDCT is presented as a system that is primarily used during a cybersecurity scenario exercise. During the scenario execution, users are provided with dialogues to annotate specific packets for tasks such as labeling. These annotations can also be used to generate rules such that similar packets receive the same labeling, making the labeling task more efficient.

Our research presented here is distinguished from these related works in several ways. First, our guidelines and framework do not explicitly focus on injecting attacks into existing source data. Our implementation, however, complements both FLAME and ID2T such that one could use both tools as part of a pipeline for NID dataset creation within the framework. Similarly, one could incorporate the INSecS-DCS tool for feature creation within our framework. We currently incorporate both Zeek[11] and Argus[12] for our container environment, however, the intention is to grow the environment such that tools like these can be incorporated based on community demand. While NDCT focuses on supporting the live annotation of network data during scenario execution, our framework would support downstream feature engineering on the resulting source data. The other main differentiator for our work compared to those discussed here, is that we focus on

[11]https://zeek.org/
[12]https://openargus.org/

being able to reproduce a dataset from source files and facilitate its exchange between researchers. The previous works in this space do not generally have this focus as they seek to improve the actual NID data itself as opposed to the process of creating it.

7.2 NID Dataset Limitations

7.2.1 Reproducibility

The main focus of the guidelines and framework presented in our work is to increase the ability of researchers to reproduce datasets from source files. To be clear, we are not concerned with reproducing and re-executing a NID scenario. Rather, we would like to take the resulting PCAP and NetFlow files from such a scenario and be able to reliably reproduce the dataset's original features and then perform further feature engineering.

An example of the need for this type of reproducibility has been researched recently by analyzing the usage of publicly available datasets by downstream researchers [24]. In most instances, the original datasets are augmented in some way, however, it was found that most of the work related to these augmentations was unable to be duplicated due to insufficient code, documentation, or both [24]. The framework delivered in our work seeks to improve this situation by providing a standard way to document and code dataset augmentations from source files.

Other work has proposed a set of content and process requirements for generating a reproducible dataset [32]. The content requirements outlined include providing full PCAP files with their data payload, anonymization of network traffic, providing ground truth data, using up-to-date network traffic, labeling the data, and providing information regarding encryption. The process requirements per-

tain to information that should be provided in order to make generating the dataset reproducible. Our guidelines support these requirements, and we look to extend them with our framework through inclusion of the scripts used to generate features and perform labeling, along with full PCAP files. This leaves no ambiguity in descriptions for how to regenerate a dataset.

In addition to these works, there is no shortage of NID literature that discusses the need to have reproducible datasets [27, 59, 93, 49].

7.2.2 Unclear Labeling Criteria

One of the major challenges that researchers face when working with NID datasets is the lack of datasets with complete and accurate labeling [59, 72]. Many of these labeling issues arise as the task is often performed by human analysis, making it both time-consuming and error-prone [59]. In other cases, the labeling criteria used is either incomplete or influenced negatively by previous errors in the dataset generation process [58].

For these reasons, accurate labels along with truth data is generally considered a major component of a useful NID dataset [27, 80, 52]. While ground truth on its own is useful, it can be misleading depending on the granularity in which it is provided. For instance, given only IP addresses and timestamps one would have to assume that any traffic related to that IP address is malicious, however, it is typically the case that a mixture of both benign and attack traffic would be present. For this reason, our proposed guidelines and framework call for the inclusion of labeling scripts which may make use of ground truth data if necessary. We note that in the case of manually labeled data this scripting could simply be index-based using the ordering of the data being processed.

7.2.3 No Standard Feature Set

In a recent series of papers, Sarhan et al. explores limitations of current datasets and the impact these limitations have on evaluating methods across multiple networks and transitioning research into practical applications [88, 89, 91, 90]. The main limitation explored in these works is the fact that with such varied features included with delivered datasets (See Table 26 in Appendix A.4), one cannot reliably compare a methodology across multiple networks to test for generalizability. This leads to hindrances during the transition from research to practical applications.

Our work looks to extend the ideas expressed by Sarhan et al. in order to enable researchers to overcome these identified limitations. We aim to make it easier for researchers to provide a dataset that is reproducible from source data and easily expanded or adjusted. In this way, one could easily use a standard feature set as well as research augmenting such a feature set for improvements.

While not the main focus, both [52] and [60] discuss and tackle the need to use a common feature set for comparison of their methods as opposed to using proprietary features delivered with most NID datasets. Both works provide informative descriptions regarding the features used in their research. In addition, [60] provides the actual calculations for the features used in their work. We believe this is a step in the right direction for the level of detail necessary to reproduce datasets from source data. We seek to naturally extend this information into scripts that are provided along with source network captures to make reproducing and extending the dataset more accessible and leave less opportunity for error.

7.3 The Intrinsic Value in NID Datasets

We believe it is worthwhile to provide a brief discussion regarding the intrinsic value provided by NID datasets as related to their development and subsequent distribution. Namely, the intrinsic value of a NID dataset is created during the scenario development, execution, and *source data* collection and not by the final delivered features. To be clear, the final features are valuable, but they are representative of a separate feature engineering activity that takes place after the intrinsic value of a network scenario has been captured in source data. In other words, the value provided by the NID dataset is derived from the actual network intrusion scenario and its collected source data. A researcher could provide any number of derived features with varying degrees of value for attack detection, however, the intrinsic value of the source data remains constant as it is derived from the scenario that was captured.

One goal of our framework is to highlight these two separate activities by advocating for the delivery of both source data and separate scripts that generate the features that take place during any subsequent feature engineering. Providing both items delivers the value of both activities to downstream researchers.

7.4 NID Dataset Delivery Guidelines

The main ideas behind the proposed guidelines are simple in statement but oftentimes overlooked in practice. Specifically considering the hand off of datasets from one researcher to another; the guidelines focus on ease of access, reproducibility from source data, verification, and extension. The guidelines are meant to provide general guidance for making the delivery of NID datasets meet these four areas of focus and reduce the impact of the limitations discussed in Section 7.2. We note

that our framework allows for the specific implementation of the guidelines to vary depending on the particular methods employed by researchers. While some common tools are provided in our framework environment, we expect it to expand to meet researchers' needs based on demand. In addition, it is important to make the distinction that when we reference reproducibility of a dataset, we refer to reproducing the dataset's final features from the original source data as opposed to recreating and re-executing the dataset's NID scenario.

The ten guidelines are outlined and described in Table 25 along with their justification and details regarding how NID researchers can implement each guideline, with a focus on our companion framework. Guidelines one through four pertain to providing downstream researchers with the resources necessary to actively reproduce and enhance the provided dataset. Guidelines five through nine outline steps that can be taken to ensure that all the dataset features and labels can be regenerated from source data, and that the steps for this generation of features can be verified and understood by downstream researchers. Finally, guideline ten is specifically included to emphasize that the delivered datasets can be considered active projects and adjust over time for any errors found after initial presentation to researchers. This aims to help avoid situations such as with the KDD Cup '99 [1] and CICIDS2017 [94] datasets, where researchers have found issues with the original datasets resulting in multiple variants of datasets being available with specific corrections [101, 58, 31].

7.5 NID Dataset Framework

In this section we cover the main ideas of our containerized environment and implementation of the guidelines. As an example of the implementation, we devel-

oped a demo dataset which takes a single PCAP file from the UNSW-NB15 dataset [73] and duplicates most of the original dataset's features and extends them to contain new features. For brevity, many specifics regarding the framework's usage have been omitted. For additional details we recommend consulting the framework repository.

7.5.1 Container Environment

We provide a containerized environment to support our implementation in order to improve reproducibility and eliminate the need to install multiple tools used by other researchers. Currently, this minimal environment includes the Zeek and Argus network analysis tools, as well as python[13] and a set of default python libraries as described in the tool's repository. It is expected that this would grow in the future, however, we consider this an adequate starting point to demonstrate its usefulness.

The intention of our framework is that the tool and our container would be used in conjunction together, however, the container environment could be used on its own just to ensure specific versions of tools are easily accessible. Running the container without specifying a command to execute will place the user into a shell prompt with access to the installed tools. The intended method of executing the environment, however, is to map the container's disk drive $/niddf$ to the directory of the user's local repository of our tool infrastructure. This allows for the development of a dataset using the framework and container to be performed in a variety of ways.

[13] https://www.python.org/

7.5.2 Framework Implementation

Our implementation provides a standard format for defining and delivering NID datasets using configuration files, naming conventions, and a standard directory structure. At the core of the implementation we read in a YAML configuration file customized for a dataset and use that information to fully process the dataset from source. The high level algorithm followed by the tool can be seen in Algorithm 2.

Algorithm 2 General processing used to generate a NID dataset based on an input configuration file. The input file is processed in a top-down manner with a loop for processing multiple source files prior to combining them together at the end.

Input: $config$, YAML configuration file
Output: $dataset$, NID dataset suitable for ML

```
Read in config

Store documentation information from config
Process setup options

Read in metadata for source data
if download_source == TRUE then
    Download all source PCAP and NetFlow files
end if

for each source file do
    Execute feature processing commands
    Execute label processing commands
    Execute post-processing commands
    Save intermediary dataset file
end for

Execute final dataset processing commands
Combine intermediary dataset files
return dataset
```

7.5.3 Dataset Directory Structure

Each NID dataset has its configuration and generation scripts contained in a dedicated directory. This allows it to be maintained by the original dataset developers and then plugged into the framework by consumers of the dataset. The general structure of a dataset directory is shown in Figure 36 where one can see the YAML configuration file, directories for source metadata, ground truth metadata, output files, and each processing step's files.

For the source and ground truth data, the directory contains metadata files which are in a comma-separated format where each line contains a download URL and the destination file name which is read in by the framework when acquiring source data. In addition, each processing step can contain simple files with the naming convention $\_\_load\_\_.<tool>$ where $<tool>$ is one of the framework's supported tools such as Zeek or Argus. While the particulars of how each tool behaves varies, these files have each line denote a single feature or process to run for a given tool. If applicable, an associated script with the same name as the feature it generates is contained in the same directory. In other words, users can easily identify the features being generated by reviewing the $\_\_load\_\_.<tool>$ files and the scripts that they reference. This promotes having easy to identify source code for each feature as indicated in guideline six, as well as having self-contained features as indicated in guideline seven.

7.5.4 Dataset Configuration File

Each dataset has a YAML configuration file that drives its creation. As seen in Listing 1 it contains documentation, options, and can contain a mix of built-in framework commands as well as custom commands to execute. For example, the

```
dataset/
├── config.yaml
├── source/
│   └── pcaps.meta
├── ground_truth/
│   └── gt.meta
├── output/
├── step_acquire_source_data/
│   ├── __load__.argus
│   ├── __load__.python
│   └── __load__.zeek
├── step_feature_processing/
│   ├── __load__.argus
│   ├── __load__.python
│   └── __load__.zeek
├── step_label_processing/
│   ├── __load__.argus
│   ├── __load__.python
│   └── __load__.zeek
├── step_post_processing/
│   ├── __load__.argus
│   ├── __load__.python
│   └── __load__.zeek
└── step_final_dataset_processing/
    ├── __load__.argus
    ├── __load__.python
    └── __load__.zeek
```

Figure 36: A default directory structure for a dataset within the proposed framework. Each processing step has its own directory intended to contain loading scripts for supported tools as well as any other scripts used in a given step. It should be noted that these directories are only needed if they are used for a given dataset. For instance, the framework takes care of default processing for several stages but the user has the option of customizing each stage with their own scripts.

framework takes information from the $setup\_options$ section and determines what source files to download during the $step\_acquire\_source\_data$ step. Other built-in commands such as run_zeek have default behavior requiring little setup on the user's part in the configuration file. In general, these commands look into the current step's directory and reads an associated $\_\_load\_\_.<tool>$ file. This file is then used by the framework to either generate features, labels, or perform some other intermediary processing. Aside from commands supported by the framework, user's can also specify any custom commands or scripting to execute, and they will be processed in the order they appear in the file. For these commands, users have access to a number of built-in variables that can be accessed in order to direct particulars, such as paths to source files to read in and where to place output. The main benefit of this single configuration file is that it fully self-describes how the dataset is created and provides the information needed for users to access the code used to generate features and perform labeling.

7.5.5 Benefits for NID Dataset Developers

The framework implementation provides several benefits for NID dataset developers. First, it provides enough flexibility such that there are varying degrees of buy-in for using the framework. For instance, suppose a NID dataset researcher only provides source files and ground truth data or has a previously generated dataset that they would like to incorporate into the framework with little effort. This can be achieved through the framework by generating the source file metadata files and ground truth metadata files. While minimum effort is required by the NID dataset researcher, it provides additional accessibility of the files to downstream consumers. On the other end of the spectrum, the container environment

provides tools for analyzing source data which can be taken advantage of by NID dataset researchers. This use of the container allows downstream researchers to use the same versions of the software when working with the dataset.

Another benefit for NID dataset researchers is that the framework implementation provides an organized structure to follow and self-documents how the dataset features and labels were generated from source data. When updating the dataset or expanding it, the change history of the configuration files within the framework can be inspected to track the changes provided there are no updates to the source data. Additionally, any improvements or feedback can be provided from end users back to the NID dataset researcher by lightweight updates to these configuration files.

The intent of this framework is such that no significant additional work is imposed on NID dataset developers as all the steps it encapsulates must already be performed to generate a given dataset. The emphasis of the framework and guidelines is such that these steps are simply organized in a standardized manner.

```
documentation:
  niddff: niddff/niddff:0.1

setup_options:
  dataset_name: demo_dataset
  source_data: unsw-nb15
  ground_truth_data: unsw-nb15
  clean_output_directory: True
  expected_outputs:
    - unsw_nb15_dataset.csv

  argus:
    clean: True
    arguments: -S 60 -m
    execute_ra: True

step_acquire_source_data:
  download: True

step_feature_processing:
  - run_zeek
  - run_argus
  - run_python_scripts

step_label_processing:
  - run_python_scripts

step_post_processing:
  - run_combine_features

step_final_dataset_processing:
  - run_combine_data
```

Listing 1: A sample input file consumed by our framework specifying where to obtain source data and how to process it to produce a final dataset. Options can be overridden on the command line if necessary.

7.5.6 Benefits for NID Dataset Consumers

This framework also provides benefits for downstream researchers using NID datasets. For researchers looking to simply use the original dataset as provided, there is generally no changes in workflow imposed by the framework, though they would be able to easily obtain the dataset using the download metadata. For researchers

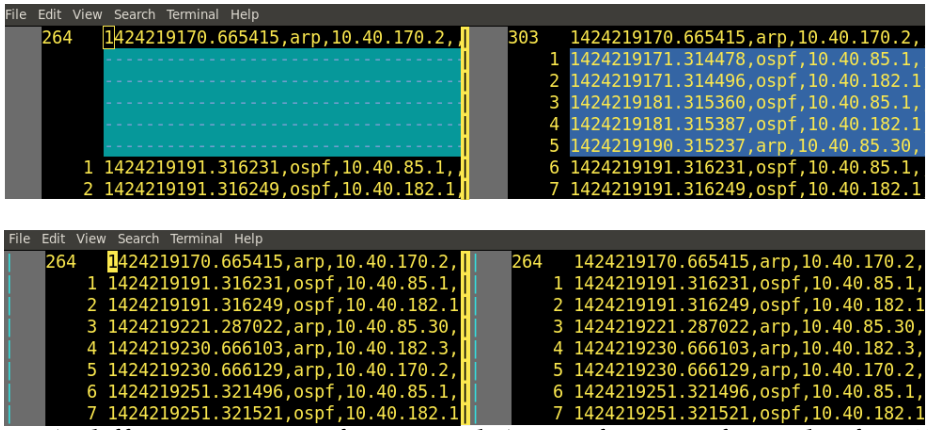


Figure 37: A diff comparison of extracted Argus features from the first PCAP of the UNSW-NB15 dataset. On the top, the left hand side of the diff shows a portion of the original Argus features from the original dataset while the right shows the same section of the output but generated by running Argus with no command line options on the source PCAP. On the bottom, the left hand side of the diff shows the same portion of the original Argus features from the original dataset while the right now shows the same section of the output generated by running Argus with the $-S\ 60$ option. The differences on the top demonstrate the necessity of having the exact command line options used to generate dataset features in order to make a dataset reproducible.

seeking to analyze a dataset, the container environment and configuration files approach provides a way for them to reproduce the dataset reliably since all the tools and the command line options used to run them are contained within the scripts. As an example of this benefit, we look at the implementation of our demo dataset, which uses a single PCAP from the UNSW-NB15 dataset [73]. As depicted in Figure 37, without using a particular set of options for Argus, one would receive results with an additional 2,529 rows compared to what the original dataset authors intended. This was found experimentally for our research but shows the value of the ambiguity that is removed when researchers have the full commands readily available. Similar benefits are gained by having the full labeling criteria laid out in the dataset configuration files.

An additional benefit comes in the form of being able to generate a standard

feature set from any source data. If some standard feature set is not included by the original dataset authors, a researcher can easily adapt the original dataset with a standard feature set in order to facilitate comparisons across multiple datasets. By following the guidelines and using the framework, the scripts to produce such a feature set become plug-n-play for any dataset that uses the same source format.

Similar to this plug-n-play nature of scripts when using the containerized environment and framework, a similar benefit can be realized for individual features of a dataset. As an example, one can consider the situation where two researchers are using the same container version and source dataset and perform different feature engineering. The use of the container and framework allows them to exchange their feature scripts or just the resulting data for *individual features* and simply merge the results into their work. This ability provides additional benefits if the environment is expanded to include a server-based component in future research.

Table 25: Guidelines for improving the handoff of NID datasets from dataset researchers to downstream researchers.

Guideline	Justification	Implementation Details
(1) Provide direct access to all data and scripts for dataset	The main purpose of this guideline is to prevent barriers to obtaining datasets. [84, 27]	This can be achieved through a simple download script. The implemented framework provides a mechanism such that dataset developers can provide metadata consisting of a download URL and destination file name to meet this guideline.
(2) Include complete source data to the most detailed extent possible	Full source data is necessary to adequately reproduce and/or augment a dataset. [84, 27, 32]	This should generally be a standard format such as PCAP or NetFlow. Full PCAP files are more favorable than partial PCAP files with no payload. If only NetFlow data is available, a full collection of attributes is better than a partial collection.
(3) If possible, provide access to all tools needed to generate dataset	Differences in tools, environments, and their versions can limit the ability of downstream researchers to obtain the same results as intended by the original dataset authors. Without this, extending the dataset with feature engineering may not be successful. [24, 91, 19]	The implemented framework meets this guideline by providing a containerized environment with specific versions of tools such as Zeek and Argus. This ensures that users of the framework can use the same baseline of tools and environment as was used by the original dataset developers.
(4) Provide documentation indicating how to reproduce a dataset from source data	Clear documentation reduces ambiguity provided in general descriptions of dataset creation. Differences in commands used to generate a dataset from source can produce different results than the original dataset. [24, 32]	Versions of tools and specific commands used to execute them should be documented. The provided framework is self-documenting as researchers can review YAML files for each dataset to view the commands used to generate them as discussed in Section 7.5.

(Table 25 Continued)

Guideline	Justification	Implementation Details
(5) Include source code needed to reproduce dataset features	Providing feature generation source code ensures downstream researchers can duplicate a dataset, verify feature correctness, and understand details of the feature calculation. [32, 84]	One should avoid making code too specific to a particular user environment. The implemented framework supports this guideline with a containerized environment, specific directories for feature generation scripts, and infrastructure to support features generated with network analysis tools.
(6) The source code for each feature should be easily identifiable	This guideline is recommended to make analysis of the features of a dataset more accessible for downstream researchers. [58, 24]	This can be implemented through naming conventions for scripts that match the final feature name and techniques such as using a separate script or function for each feature. The implemented framework supports this by enforcing these conventions in its interfaces with network analysis tools.
(7) The generation of each feature should be independent from others	This guideline is recommended to avoid execution dependencies between features and it facilitates the ability to remove or add new features by downstream researchers. This also makes the code for each feature more understandable and reviewable. [58, 24]	The implemented framework supports this guideline in the way it interfaces with network analysis tools to generate features in an independent manner where possible. In addition, the built-in framework encourages this by providing standard configuration files that can be used to identify each feature script to run.

(Table 25 Continued)

Guideline	Justification	Implementation Details
(8) Apply guidelines outlined for features to labels as well	While labels are significant for model training, during dataset generation time, they can be considered a special case of features. In this way, we want to apply guidelines (4), (5), and (6) to labels as well. [32, 59, 27, 80, 52]	The implemented framework supports this goal by providing the same infrastructure available for feature development to label development.
(9) Make source code for labeling distinct from other features	This guideline is recommended to make the labeling criteria used for a dataset clear for collaborating researchers. Because the label features/procedure can inform machine learning model design decisions it is helpful to have it distinctly identifiable. For example, if the labeling criteria is based on a single IP address, it is likely that the IP address features should not be provided to a model. [58]	The implemented framework supports this guideline by having a separate step of processing for label scripts and by having them contained in a separate directory for a given dataset.
(10) Provide a mechanism to receive and implement feedback from researchers to correct issues and improve dataset	This guideline encourages collaboration between NID researchers, allows a dataset to remain current, and provides a feedback loop to dataset researchers to correct any issues found by the research community. [84, 58]	The implemented framework supports this guideline through its use of scripting and metadata to describe a dataset such that each dataset can be maintained in an independent source code repository or as part of the default environment.

8 Conclusion

In this dissertation we presented several methodologies for improving network intrusion detection using deep learning with autoencoders. We first showed the application of autoencoder dimension estimation as a suitable methodology for network attacks that have network-wide footprints. We then developed the generation of novel feature sets using autoencoder feature residuals and explored pairing them with a variety downstream classifiers for performing network intrusion detection. Ultimately, this work showed that using feature sets generated with autoencoder feature residuals improves classification performance over an original feature set for network intrusion detection using a robust number of cybersecurity scenarios and attacks. In other words, autoencoder feature residuals can be used as a drop-in replacement for an original feature set with little risk of degrading classifier performance while often providing an increase in performance.

We then provided a focus on practical matters related to the hand off of network intrusion detection datasets between researchers. A set of ten guidelines were developed and outlined in detail along with an implemented framework that can be used by both dataset researchers and consumers to improve the reproducibility of feature engineering performed on source data.

Through a blend of both theoretical and practical methodology, we believe that this research has helped improve our ability to perform network intrusion detection and has laid the groundwork for future exploration using autoencoder feature residuals.

9 Future Work

We believe that the work presented in this dissertation provides the opportunity for extension through further research. Suggestions for interesting areas of extension are listed below.

- Further research could be performed to determine the effectiveness of various autoencoder architectures at generating autoencoder feature residuals. Examples could include variational autoencoders, sparse autoencoders, and denoising autoencoders.

- In this research we generally focused on the effectiveness of using only benign NetFlow data to train our autoencoders. It would be interesting to explore the effects of using varying amounts of attack NetFlow data during training to help drive differences in the autoencoder feature residuals of benign samples compared to attack samples.

- This work primarily focuses on the applicability of autoencoder feature residuals in the domain of network intrusion detection. An intriguing extension would be exploring other domains to research if their effectiveness is universal.

- Excepting the one-class classifiers explored, many of the downstream classifiers explored were supervised methods, which limits their suitability for anomaly detection. It would be exciting to explore combining autoencoder feature residuals with additional unsupervised and single class techniques such as a one-class neural network [21].

- While we explored using recurrent methods such as RNNs and LSTMs as downstream classifiers, a compelling extension of this technique would be to incorporate recurrent layers in the autoencoder. Doing so has the potential to account for the sequential nature of network activity in the autoencoder feature residuals.

A Appendix

A.1 Network Intrusion Detection Using Iterative Neural Networks and Autoencoder Feature Residuals

In this section we outline the final work contributing to this dissertation which was in progress at the time of this writing. We have included our current results and their analysis so far, but note that this area could be enriched further and benefit from additional experimentation. Here, we explore concepts developed by Paffenroth and Hershey, and initially explored in Hershey's thesis [43]. We seek to exploit the sequential nature of network data to detect network attacks using an alternative formulation of neural networks, and recurrent neural networks in particular, referred to as iterative neural networks.

A.1.1 Overview

Traditionally neural networks are defined through the expression of layers, nodes, and non-linear activation functions. In doing so, we generally formulate a parameterized network such that we can approximate some true function using training data, which is assumed to be representative of our true problem. A tedious amount of time and hyperparameter tuning often goes into determining the proper architecture for neural networks such as MLPs, RNNs, and LSTMs.

This tuning is often performed under the assumption that the actual structure of the network, in terms of layers and nodes, is an optimal way to construct neural networks. In other words, we often assume that an exhaustive number of neural network representations have been explored and this logical representation has been deemed ideal. Drawing on influences from dynamical systems, iterative neu-

ral networks (INNs) challenge this notion by offering an alternative way to formulate neural networks in terms of a weight space of trainable parameters such that the parameters are applied recursively against the network's input over multiple repetitions [43]. We next briefly formulate INNs to make their structure and definition clear, followed by an examination of our current results exploring their application to network intrusion detection.

A.1.2 Iterative Neural Networks

Our aim with defining INNs is to show that it is a broader class of neural networks which is flexible enough to be equivalent to MLPs and RNNs [43]. To do this, we start with our definition of an MLP, and show how it can be expressed in an iterative manner. MLPs with an arbitrary number of layers as outlined in Section 2.4 can be expressed using Equation 15. For simplicity, we start with an MLP of only two layers which can be expressed using Equation 56.

$$f(\theta, x) = g(W_2 g(W_1^T x + c_1) + c_2) \tag{56}$$

This can then be to be simplified into matrix notation as in Equation 57

$$f(X) = g(W^2 X g(W^1 X)) \tag{57}$$

by having X be a d x n matrix of all n input samples with each sample having d dimensions. Adding a single vector of ones to X allows us to concatenate the biases into our weight matrix W. In this way each layer i can be expressed using Equation 58

$$f^i(X) = g(W^i X) = H^i \tag{58}$$

where H^i is the output of a given hidden layer, with the final layer returning our network's final output. Returning to our two layer MLP, it can be represented in this new form as shown in Equation 59 [43]

$$f^1 \circ f^0(X) = g(W^1(g(W^0 X)) = Y \tag{59}$$

where Y is the output of our network.

We now use these matrices to create our INN using a square matrix $f(X)$ with the transition from MLP to INN shown in Figure 38. In this figure one can see that we vertically concatenate the component matrices X, H, and Y from the MLP vertically to serve as a single input matrix $\bar{X}$. Both H and Y in $\bar{X}$ are initialized as zero matrices with the dimensions of $\bar{X}$ being $(d + h + y)$ x n. The iterative weight matrix $f(X)$ is a square matrix with dimensions $(d + h + y)$ x $(d + h + y)$ as shown in the middle portion of Figure 38. We replicate the two layer MLP in an iterative fashion by taking $f(X)$ and multiplying it twice by $\bar{X}$ where each iteration represents a layer in the original MLP. This iterative function can be expressed using Equation 60 [43].

$$f(f(\bar{X})) = g(W^1 g(W^0(X)) = Y \tag{60}$$

As shown in the bottom portion of Figure 38, for INNs, a more generalized representation of the bottom $(h + y)$ rows of $f(X)$ can be realized by considering the entirety of this area of the matrix to be our *weight space*. In other words, we can tune our INN using various compositions of trainable parameters and sparsity

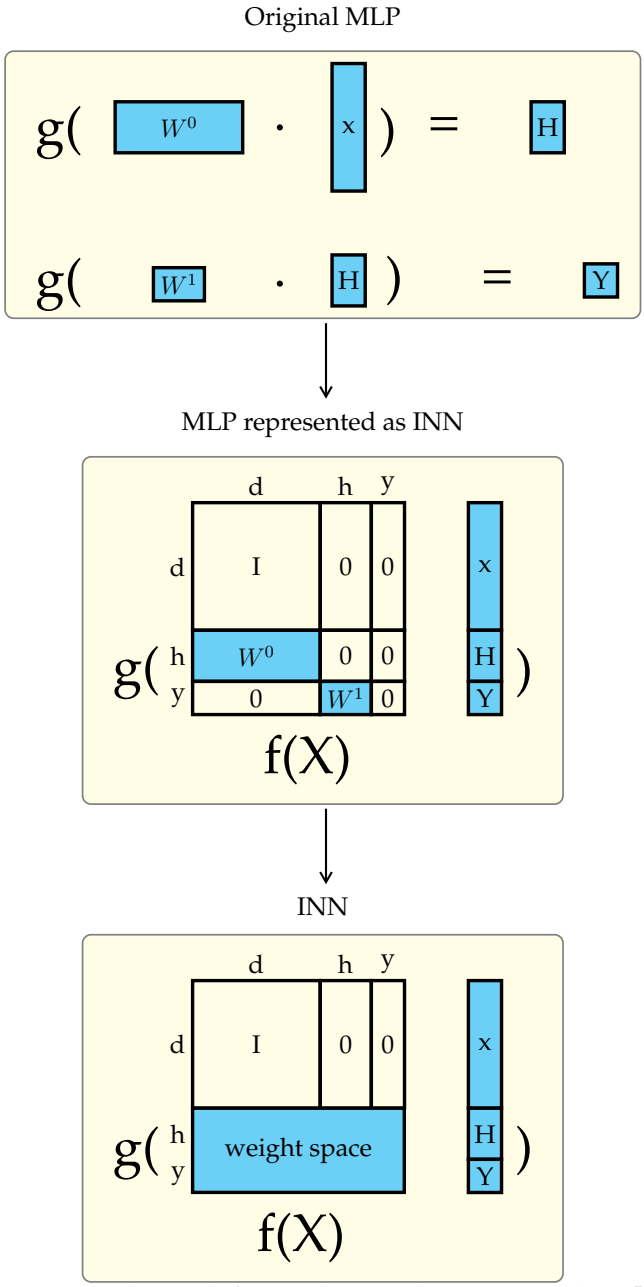


Figure 38: This figure is adapted from the work presented in [43]. Transforming a traditional two layer MLP to an equivalent INN. We start with the original two layer MLP and then represent it using a square matrix with trainable weights that can be applied iteratively to a stacked representation of X, H, and Y. We then relax the weight space to be arbitrary layouts of weights which we consider a generalized INN.

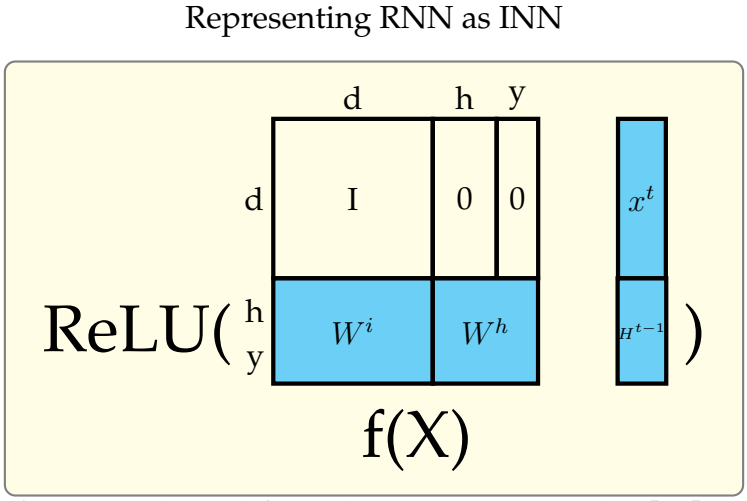


Figure 39: This figure is adapted from the work presented in [43]. The notional details regarding how an RNN can be represented using an iterative structure with INNs. Here we divide the weight space into two halves for the weights corresponding to the weights of an RNN. Additionally, we collapse H and Y together to be the hidden state of the previous time step H^{t-1}.

within this area of the iterative matrix.

A.1.3 Iterative Neural Networks for Sequential Data

Having shown the transition from an MLP to an equivalent INN, we can now relax some of the structural constraints to allow the INN structure to process sequential data in an iterative manner. To do this, we consider the weight space to be a single dense matrix of trainable parameters which can notionally be divided into two halves referred to as W^i and W^h as shown in Figure 39. Also depicted in Figure 39, we collapse Y and H together to form H^{t-1} and use the ReLU activation function [43]. In this form, we see that we have formed an iterative equivalent to an RNN represented by Equation 61 [43].

$$H^t = ReLU(W^i X^t + W^h H^{t-1}) \tag{61}$$

We note one difference between traditional RNNs and INNs as formulated here, related to the common usage of a final dense projection layer for the output of a traditional RNN. While this can be replicated with INNs by constraining parts of the weight space to be an untrainable matrix and requiring an additional iteration, we generally do not enforce this constraint when working with INNs, favoring to allow the weight space to take on broad formulations of dense and sparse weight distributions. The implication, though generally negligible, is that the weight space must compensate for the fact that the final output Y^t does not receive the hidden state H^t at time step t because they are calculated at the same time in the INN formulation.

A.1.4 Applying Iterative Neural Networks to NID

In our application of INNs to NID, we executed preliminary experiments to compare X and S across RNNs, LSTMs, and INNs while varying the number of parameters and levels of sparsity. The experimental setup remains the same as our previous work with autoencoder feature residuals with the exception that the classifiers used the iterative structures described in Section A.1. Our data was encoded using our standard dataset processing outlined in Section 6.3 and Table 5. We note that we used stratified random sampling of sequences across entire datasets based on the samples being attack or benign network flows, which is a slightly different strategy than previous work with recurrent methods that was employed in Section 6.6.2. In our previous work, we had constrained our random sampling to the areas of datasets that contained attack scenarios. This new strategy was used as it is more practical to maintain across a large number of datasets. Each sequence consists of 25 NetFlow samples where the label for prediction is the label of the

final sample of the sequence.

Our primary comparisons used RNN and LSTM models as a baseline for comparison to INNs with varying levels of sparsity in the weight space. This sparsity was handled through a parameter referred to as R which can range from 0.0 to 1.0. The value of R indicates the likelihood that a given weight in the weight space will be trainable or not, where higher values indicate it is more likely to be trainable. In other words, as R approaches zero, the weight space becomes increasingly sparse. To maintain a certain number of parameters, the overall size of the weight space is increased to account for the fact that some weights will no longer be trainable compared to a dense INN.

Our preliminary results are currently inconclusive as for the effectiveness of this technique to NID. Examining Figures 40 and 41 for the UNSW-NB15 dataset, one can see that the LSTM performed best overall with the highest marks being achieved using S in combination with an LSTM. This boost in performance using S is in line with our previous findings in this area. Our initial expectation was that the introduction of sparsity in the INN weight space would allow the INN models to outperform the LSTM with fewer parameters, as was found in previous work using benchmark image datasets [43]. For the UNSW-NB15 dataset, however, this was not the case as the LSTM model outperformed all other methods handily regardless of the number of parameters used or sparsity level. The best performance on this dataset was achieved using S as input to an LSTM model with 24,480 parameters, which achieved an f1-score of 0.766. We also note, that an LSTM using S as its input features with only 306 parameters achieved an f1-score of 0.573. This result was better than all RNN and INN models across the parameters examined for the UNSW-NB15 dataset.

We now examine results using the ToN-IoT dataset, which run contradictory to our findings with the UNSW-NB15 dataset. In these results, shown in Figures 42 and 43, we see that introducing sparsity has caused the INN to achieve superior performance compared to an RNN and LSTM model. This is true when using either X or S as input to the models, with using S providing the best results overall, achieving an f1-score of 0.991 with an INN that contains 11,249 trainable parameters and the sparsity parameter R set to 0.7. We note that for this dataset, we were able to achieve an f1-score of 0.978 using an INN with only 346 parameters and R set to 0.3, which outperformed all LSTM models using up to 24,480 parameters. While these intial results are promising for the ToN-IoT dataset, further exploration would need to be performed in order to make definitive statements regarding the effectiveness of sparse INNs for NID.

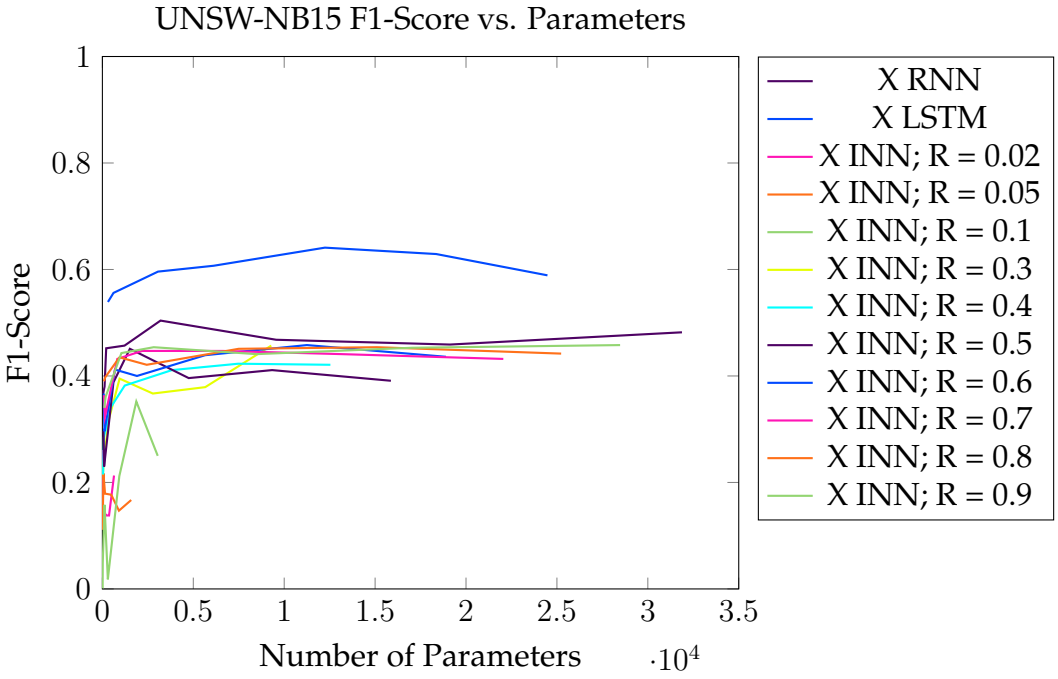


Figure 40: Here we show a comparison using X along with models formulated in an iterative manner. The parameter R indicates the amount of sparsity applied to the parameter space in the INN models where the values for R range from 0 to 1 and indicates the likelihood of a given parameter being trainable or not. This plot shows that using an LSTM performed better than all other models across all numbers of parameters.

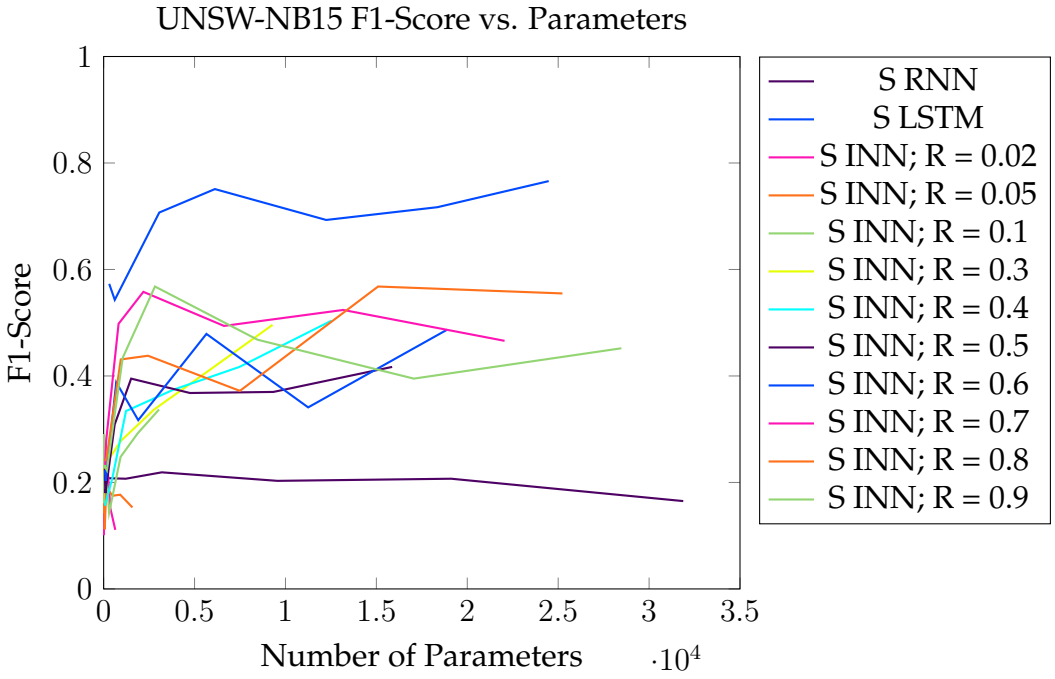


Figure 41: Here we show a comparison using S along with models formulated in an iterative manner. The parameter R indicates the amount of sparsity applied to the parameter space in the INN models where the values for R range from 0 to 1 and indicates the likelihood of a given parameter being trainable or not. This plot shows that using an LSTM performed better than all other models across all numbers of parameters.

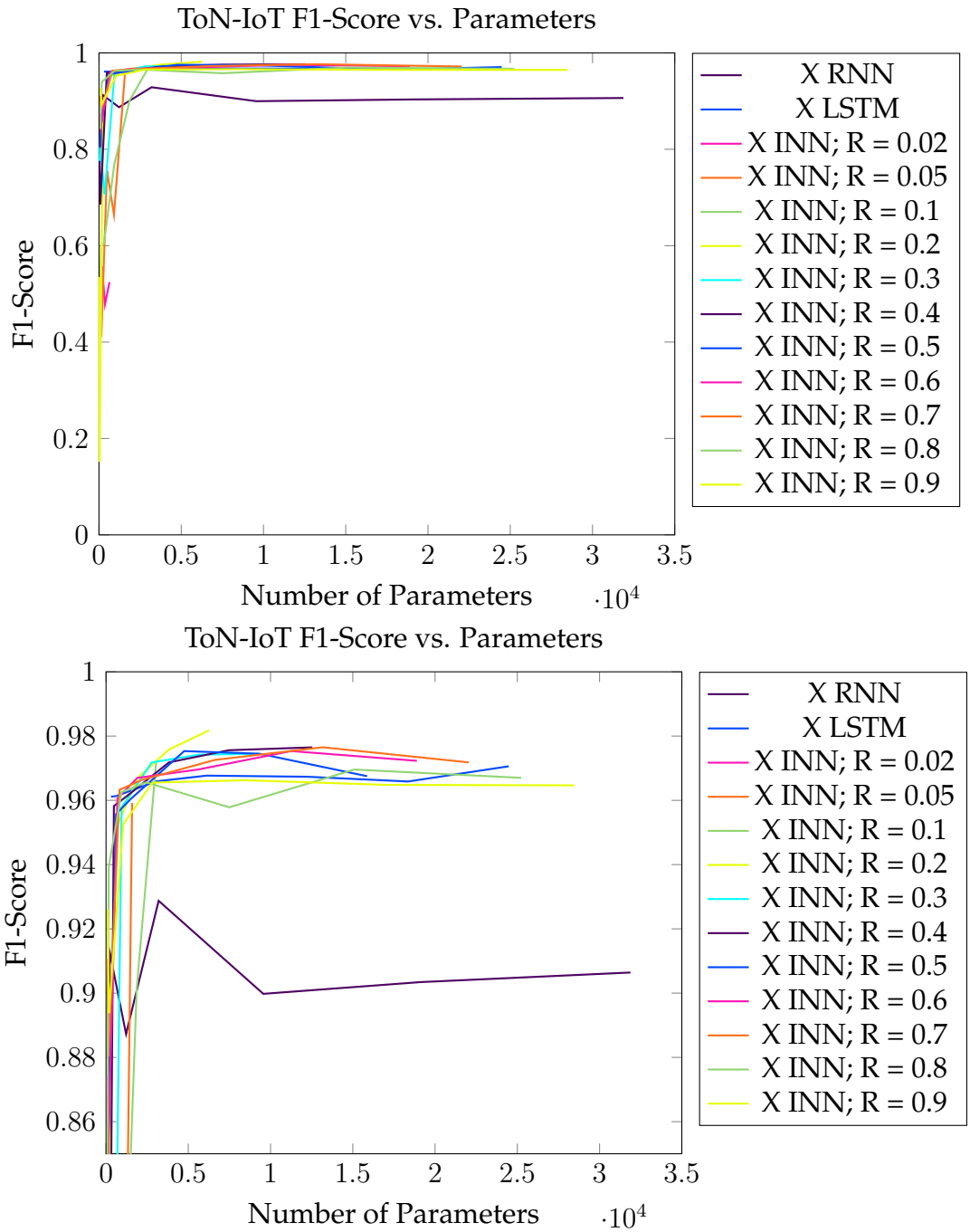


Figure 42: Our results using the ToN-IoT dataset as input to INNs using only X as features. The parameter R indicates the amount of sparsity applied to the parameter space in the INN models where the values for R range from 0 to 1 and indicates the likelihood of a given parameter being trainable or not. The bottom plot shows a smaller range compared to the to plot to make additional details visible. Here one can see that both an LSTM and the sparse INNs outperformed the RNN models. Additionally, the sparse INNs achieved the highest performance overall.

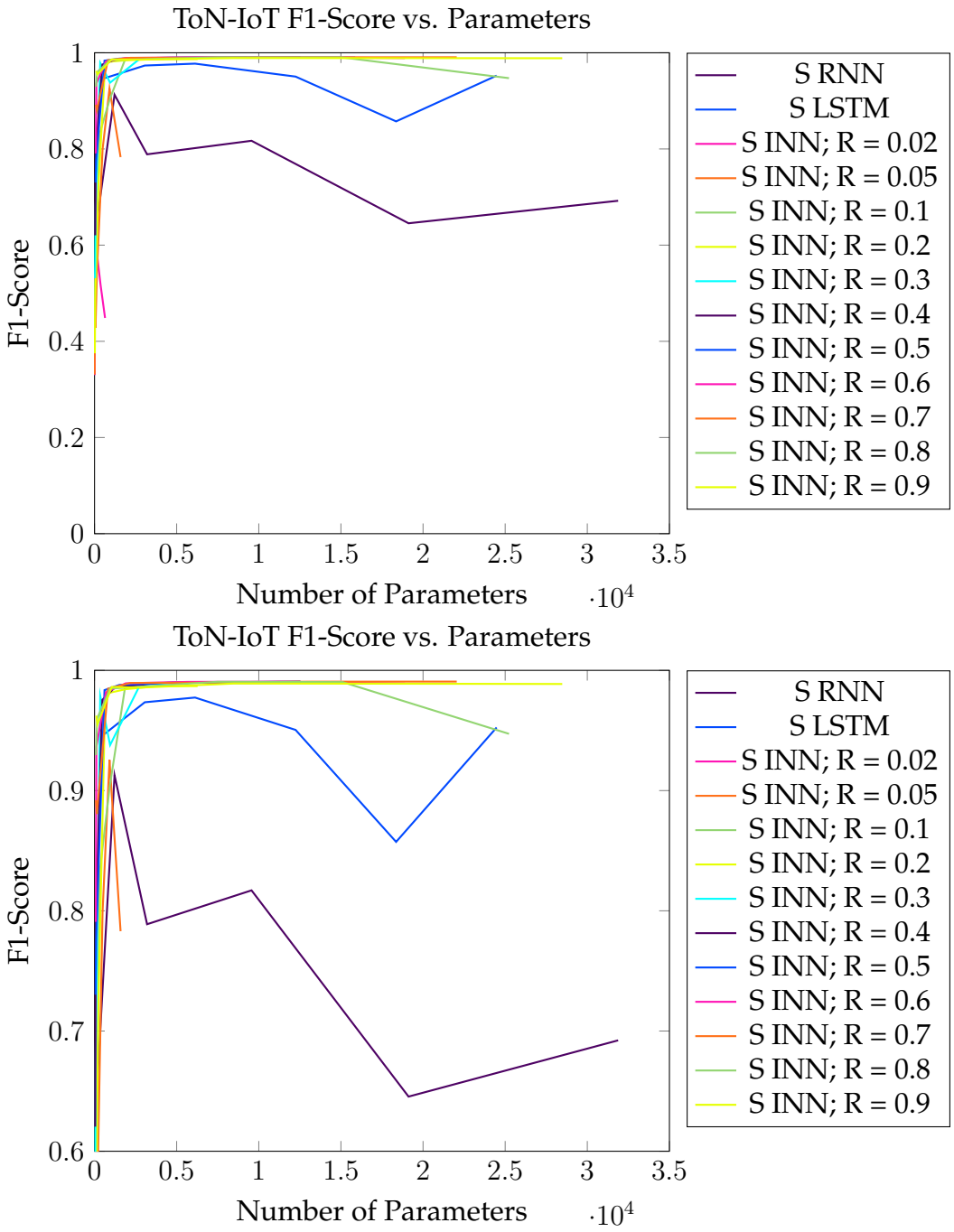


Figure 43: Our results using the ToN-IoT dataset as input to INNs using S as input features. The parameter R indicates the amount of sparsity applied to the parameter space in the INN models where the values for R range from 0 to 1 and indicates the likelihood of a given parameter being trainable or not. The bottom plot shows a smaller range compared to the to plot to make additional details visible. Here one can see that both an LSTM and the sparse INNs outperformed the RNN models. Additionally, the sparse INNs achieved the highest performance overall.

A.2 Performance Metrics

In this section of the appendix we provide details regarding the performance metrics used throughout our research. As the majority of the metrics are calculated in relation to classifying network attacks we outline some initial common language used when calculating performance metrics.

- **true positive (TP)**

  We consider a true positive to be when we classify a sample as an attack and the true label of that sample is an attack.

- **true negative (TN)**

  We consider a true negative to be when we classify a sample as benign and the true label of that sample is benign.

- **false positive (FP)**

  We consider a false positive to be when we classify a sample as an attack, however, the true label of that sample is benign.

- **false negative (FN)**

  We consider a false negative to be when we classify a sample as benign, however, the true label of that sample is attack.

A.2.1 Accuracy

Accuracy is a simple measure that provides the proportion of samples for which the classifier produces the correct prediction and can be calculated using equation 62 [39].

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{62}$$

While this can be a suitable metric for many problems, it is not well suited for heavily imbalanced problems such as network intrusion detection. In network intrusion detection, in most cases one could simply always predict samples to be benign and achieve a high accuracy. For this reason, we general rely on precision, recall, and f1-score which are discussed in the following section.

A.2.2 Precision

Precision is a metric that reports the fraction of positive classifications that are correct out of all positive predictions made [39]. In our case, this means it is the number of samples we classify as attacks that are also labeled as attacks when considering all samples that we classified as attacks. This is expressed in equation 63.

$$\text{precision} = \frac{TP}{TP + FP} \tag{63}$$

Values for this metric range between zero and one with larger values indicating better performance.

A.2.3 Recall

Recall is a metric that reports the fraction of true positive classifications that were detected out of all possible true positive events that occurred [39]. In our case, this means it is the number of samples labeled as attack that we classified correctly compared to all of the samples that were labeled as attack. This is expressed in equation 64.

$$\text{recall} = \frac{TP}{TP + FN} \tag{64}$$

Values for this metric range between zero and one with larger values indicating better performance.

A.2.4 F1-Score

Classification performance can be considered a trade-off between both precision and recall as defined in sections A.2.2 and A.2.3. For this reason we often report the f1-score, which is the harmonic mean between precision and recall [39]. This measure can be expressed as shown in equation 65.

$$\text{f1-score} = \frac{2 * precision * recall}{precision + recall} \tag{65}$$

Values for this metric range between zero and one with larger values indicating better performance. We note that this metric remains dependent on the ratio of positive to negative samples being used. However, in our work we are generally comparing performance using different feature sets using the same datasets. As the ratios of attack and benign samples remain the same when doing this, we can use the metric effectively for comparative performance reporting.

A.2.5 False Alarm Rate

The false alarm rate provides the ratio of false positives compared to the total number of negative samples presented to a classifier [46]. In our work this is the number of samples we classify as attacks that are truly benign compared to all benign samples. This can be expressed as shown in equation 66.

$$\text{False Alarm Rate} = \frac{FP}{FP + TN} \tag{66}$$

This metric ranges from zero to one where values closer to zero indicate better performance. In the field of network intrusion detection we seek to minimize the false positive rate as false positives often result in alerts to network operators and ultimately wasted resources.

A.3 Common Equations

A.3.1 Min-Max Normalization

The goal of min-max normalization is such that each feature has a value between zero and one. As shown in equation 67

$$\hat{x}_k = \frac{x_k - min(x_k)}{max(x_k) - min(x_k)} \tag{67}$$

where x_k is the kth feature. This scaling is determined using training data and then applied to training, validation, and test data. This can result in some values greater than one or less than zero if the original value exceeds the min and max values of the training data.

A.3.2 Mean Squared Error (MSE)

We use the mean squared error to calculate the loss for our autoencoder using the general equation 68

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y - \hat{Y})^2 \tag{68}$$

where Y is the target value and $\hat{Y}$ is the predicted value. In our case we are using an original input X and an autoencoder reconstruction yielding equation 69

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(X - D(E(X)))^2 \tag{69}$$

A.3.3 Two Sample Kolmogorov-Smirnov Test (KS-Test)

To support comparisons between our mean f1-scores we perform a two-sample Kolmogorov-Smirnov test (KS test). The two-sample KS test takes two samples and provides a statistic defined by equation 70

$$D = \underset{u}{sup}|F_m(u) - G_n(u)| \tag{70}$$

where F_m is the empirical distribution function of one sample containing m data points and G_n is the empirical distribution function of a sample containing n data points [45]. We have u as the union of both samples used for comparison where the statistic returned is the supremum of the absolute value of the differences between the two distributions across u [45]. In our case, we have F_m be the empirical distribution function for results using X as input features and G_n is our method's results. In our case $m = n$ as we take the same number of runs for each experiment for comparison. The KS test statistic is used to determine if the two input samples are from the same distribution [45]. We use associated p-values for this purpose with a p-value less than or equal to 0.05 indicating we could reject the null hypothesis that the two samples come from the same distribution.

A.4 Dataset Descriptions

Here we provide descriptions and references for datasets used in this dissertation as well as some of the popular datasets that are commonly found used in research. A high level summary of the features of the datasets can be seen in Table 26.

Table 26: Overview of some of the widely used network intrusion detection datasets. In our research we concentrated on datasets that include labels, provide access to network flow samples of the data, and are publicly available.

Dataset	Year	# Features	Real vs. Synthetic	Attacks
KDD99	1999	41	Synthetic	DoS, U2R, R2L, Probing
NSL-KDD	2009	41	Synthetic	DoS, U2R, R2L, Probing
CTU-13	2014	12	Mixed	IRC, Spam, ClickFraud, Port scan, DDoS, FastFlux, P2P, HTTP
UNSW-NB15	2015	49	Real benign; Synthetic attack	Fuzzers, Analysis, Backdoors, DoS, Exploits, Generic, Reconnaissance, Shellcode, Worms
WSN-DS	2016	23	Synthetic using NS-2	Blackhole, Grayhole, Flooding, Scheduling
UGR16	2016	12	Real benign; Synthetic attack	Low-rate DoS, Port scan, Botnet
CIDDS-001	2017	11	Synthetic using OpenStack	Ping scan, Port scan, Brute force, DoS
CIDDS-002	2017	11	Synthetic using OpenStack	Port scan, SSH Brute force, DoS
CICIDS2017	2017	80	Synthetic; Using machine profiles	Brute force, Heartbleed, Botnet, DoS, DDoS, Web, Infiltration, Port scan
CSE-CIC-IDS-2018	2018	80	Synthetic; Using machine profiles	Brute force, DoS, Web, Infiltration, Botnet, DDoS, Port scan
BoT-IoT	2019	29	Synthetic in virtual environment	Port scan, OS fingerprinting, DoS, Data theft, Keylogging
SDN-IoT	2020	31	Synthetic	Dos, DDoS, Port scan, OS/Service fingerprinting, Fuzzing
IoT-23	2020	21	Real (lab-based)	Dos, DDoS, Port scan, botnet, file download
ToN-IoT	2021	44	Synthetic using NSX vCloud NFV platform	Scanning, DoS, DDoS, Ransomware, Backdoor, Injection, XSS, Password cracking, MITM
NF-UNSW-NB15-V2	2021	43	From UNSW-NB15	See UNSW-NB15
NF-BoT-IoT-V2	2021	43	From BoT-IoT	See BoT-IoT
NF-ToN-IoT-V2	2021	43	From ToN-IoT	See ToN-IoT
NF-CSE-CIC-IDS-2018-V2	2021	43	From CSE-CIC-IDS-2018	See CSE-CIC-IDS-2018

A.4.1 KDD99

The KDD99 dataset captures seven weeks of data from the DARPA '98 evaluation program [101]. At the time of its release it was one of the most robust datasets available and included properties such as a dedicated test set with attacks not contained in the training set [101]. While we cover this dataset for historic purposes, there have been a number of corrected issues with the dataset such that one should favor using the NSL-KDD dataset [101]. The specific attacks in this dataset are now outdated and research using this as a benchmark should be looked at with a discriminating eye.

A.4.2 NSL-KDD

After the release of the KDD99 dataset, a number of researchers found issues pertaining to inconsistencies in attack definitions, duplicate records, and its data distribution [101]. In response to these issues, the authors of [101] resolved these issues and produced the NSL-KDD dataset. After doing so, the usable size of the KDD99 dataset was reduced by 75.15% [101]. While this dataset is now aged, it is still in wide use for benchmarking network intrusion detection techniques. Some drawbacks to this dataset include that it has a custom feature set and is now outdated. While it is preferred over the KDD99 dataset and useful for benchmark comparisons with other methods due to its widespread use, it is preferred to use other datasets to fully assess new techniques.

A.4.3 CTU-13

The CTU-13 dataset consists of 13 botnet scenarios captured using the Czech Technical University network [34]. This dataset combines real botnet traffic along with

a mixture of normal and background traffic. The dataset provides both packet capture (PCAP) files and NetFlow data generated using Argus. As is typical for most cybersecurity datasets, there is a large imbalance between the number of attack and benign samples as would be typical in most real world network settings.

A.4.4 UNSW-NB15

The UNSW-NB15 dataset was developed by the University of South Wales in order to improve upon existing datasets by using real modern network traffic and the low network footprint of modern-day attacks [73]. With the goal of helping to improve network intrusion detection systems, the dataset contains nine different attack types along with benign network traffic. Interestingly, the dataset is constructed using real modern benign network traffic combined with synthetic network attacks. The data is available in a labeled CSV format containing NetFlow data and features generated using the Argus and Bro network tools.

A.4.5 WSN-DS

The WSN-DS dataset was produced specifically to improve research on network intrusion detection for wireless sensor networks [6]. The dataset focuses on denial of service attacks for which wireless sensor networks may struggle against given their generally lower processing power compared to traditional networks [6]. In addition, the dataset is used to assess how the LEACH protocol used in wireless sensor networks performs in the presence of these attacks [6]. While this dataset is not seen as often as other benchmarks, it is an example of a network intrusion detection dataset with a specific focus for wireless sensor networks and the LEACH routing protocol in particular.

A.4.6 UGR16

The UGR16 dataset was generated by the University of Granada using real network traffic captured over the course of several months combined with synthetic attack traffic [65]. The real network traffic included was collected from a Tier 3 ISP network. One of the goals of this dataset was to capture network traffic over a long period of time in order to include the cyclostationary nature of network traffic [65]. This quality of the data may make it appropriate for researching network drift as well as RNNs.

A.4.7 CIDDS-001

The CIDDS-001 dataset was developed by researchers in order to provide an anomaly detection network intrusion dataset [83]. This dataset was captured over four weeks using a network simulated with OpenStack in combination with python scripts to emulate specific behaviors [83]. The environment emulates a small office with clients and servers and attempts to recreate network drift characteristics according to working hours. The main idea behind this dataset was to emulate insider threats that businesses may face that generate within their internal networks.

A.4.8 CIDDS-002

This dataset is an extension put together by the researchers that generated the CIDDS-001 dataset [82]. Unlike the CIDDS-001 dataset, the main focus of this dataset is on various port scanning techniques [82].

A.4.9 CICIDS2017

The CICIDS2017 dataset consists of five days of network data including one full day of benign data that was developed by the Canadian Institute for Cybersecurity at the University of New Brunswick [95]. This dataset aimed to provide up-to-date network attacks and realistic background traffic which was lacking in many available datasets at the time of its publishing. The delivered features were captured by post-processing PCAP files using the CICFlowMeter software developed by their group and also made publicly available [95]. Its scenarios have been used as a benchmark cybersecurity dataset for many works and is still in use today. Recently, researchers have provided updated versions of this dataset that address potential issues found with the dataset and should also be considered for use [58].

A.4.10 CSE-CIC-IDS-2018

The CSE-CIC-IDS-2018 dataset was developed by the Canadian Institute of Cybersecurity at the University of New Brunswick [14]. Generated using sets of profiles for normal and attack behaviors it simulates a mixture of hundreds of hosts along with servers that are attacked by a set of 50 attack hosts. Similar in nature to the CICIDS2017 dataset produced by the same research group using the same toolset, it serves to provide an updated set of attacks targeted to a larger infrastructure.

A.4.11 BoT-IoT

The BoT-IoT was developed by researchers at UNSW Sydney in Australia [53]. The main goal of this dataset was to have a representative network intrusion detection dataset consisting of IoT devices attacked in the context of a botnet [53]. To do

[14] https://www.unb.ca/cic/datasets/ids-2018.html

this a virtual environment was set up that included IoT devices such as a weather station and various smart home devices. The Ostinato tool was used to provide large amounts of benign traffic for the scenarios in this dataset.

A.4.12 SDN-IoT

The SDN-IoT dataset is a relatively new dataset being introduced in 2020 [48]. The objective of this dataset is to provide samples of an SDN controlled network with IoT devices within it. Modeled after the ToN-IoT dataset, it contains many of the same attacks and similar architecture. It includes five attacks, but the dataset as a whole is dominated by port scanning attacks [48]. While this dataset was used in [81] recently, it will be interesting to see if this becomes a common benchmark for network intrusion detection.

A.4.13 ToN-IoT

The ToN-IoT dataset was developed to provide a realistic benchmark in the Internet of Things environment [70]. To develop the dataset, researchers utilized a software defined network, network function virtualization, and service orchestration facilitated by the NSX vCloud NFV platform. Unique in this goal of targeting a specific type of network environment, it provides nine network attack types on IoT devices and infrastructure.

A.4.14 IoT-23

The IoT-23 dataset was developed in the Stratosphere Laboratory in efforts to create a large amount of real labeled malware and benign IoT traffic to be used for network intrusion detection research [35]. It consists of 23 IoT scenarios executed

in a controlled environment using real IoT devices in combination with malware executed from Raspberry Pi devices. In this work we utilize scenarios 1, 13, 19, and 20 chosen mainly for practical reasons based on the conservative number of network flows included in these scenarios. Additionally, we augment the scenarios by including all the benign samples from the three benign-only scenarios included in the dataset. We refer the reader to the original work for details of each scenario [35].

A.4.15 NF-V2 Datasets

The NF-V2 datasets are a collection of previously developed datasets restructured from their PCAP files to all have a standard set of features [91]. Motivated by standardizing benchmark features in order to facilitate comparisons among network intrusion detection research, these datasets all contain features from the NetFlow v9 protocol. There is a restructured dataset for UNSW-NB15, ToN-IoT, BoT-IoT, and CSE-CIC-IDS-2018, as well as a dataset that combines all of them together. One drawback for these datasets is that the timestamp data for network flows is not included, making them less viable when using techniques such as RNNs or focusing on the sequential aspect of network data.

A.5 NetFlow Encoding Details

A.5.1 Communication Type One Hot Encoding

Communication type is determined using the source and destination ports utilized in a given NetFlow record. In some instances, datasets only provide an entry such as “service” which we were also able to use for mapping to a communication type. To determine the actual fields appearing in Table 27 we took counts of occurrences

across many public benchmarks and utilized the most commonly used ports leaving an "other" field for any less used or unseen ports.

Table 27: Communication type one hot encoding features found using source and destination ports present in NetFlow records.

One Hot Encoding Feature	Mapped Port Numbers
system_defined	0
ftp	21
ssh	22
dns	42, 53, 135, 5353
http	80, 443
kerberos	88
ntp	123
samba	137, 138, 139
ldap	389, 3268
ms-smb	445
smtps	465
registered	ports $\geqslant$ 1024 and $\leqslant$ 49,151
user_application	ports $\geqslant$ 49,152
other	any unmappable ports

A.5.2 Protocol One Hot Encoding

The protocol one hot encoding was determined based on the most used protocols across a large swath of network intrusion detection datasets. Some datasets provide a protocol number while others use an actual protocol name which we account for in our mappings. It should be noted that we include all logical OSI layer protocols in this mapping and do not separate them. Details of the mapping can be found in Table 28.

Table 28: Protocol one hot encoding column mappings based on protocol values contained in a NetFlow record.

One Hot Encoding Feature	Mapped Values
rtp	rtp, rtcp
tcp	6, tcp
udp	17, udp, udt
sctp	132, sctp
icmp	1, icmp
igmp	2, igmp
tunneling	50, 47, 94, esp, gre, ipip, ipnip
ip	0, 41, 55, 4, 44, 43, 59, 60, ip, ipv6, ipv6-frag, ipv6-route, ipv6-no, ipv6-opts
arp	54, 91, arp
ospf	89, ospf
other	any unmappable protocol

A.6 Machine Learning Algorithm Default Parameters

In our work we often use the default parameters set by the sklearn algorithms as we are interested in comparative performance as opposed to optimizing performance. The following set of tables provide what these default parameters are based on the sklearn libraries. Unless otherwise noted, when performing ablation studies of a particular parameter, the remaining parameters would remain set to the values in these tables. For additional descriptions of what each parameter controls one can review the documentation for the algorithm for the sklearn library[15].

[15]https://scikit-learn.org/stable/modules/classes.html

Table 29: The parameters and their default values from the sklearn algorithm for naive bayes classifier.

Parameter	Type	Default Value
priors	array-like	None
var_smoothing	float	1e-9

Table 30: The parameters and their default values from the sklearn algorithm for K-Nearest Neighbors.

Parameter	Type	Default Value
n_neighbors	int	5
weights	str	uniform
algorithm	str	auto
leaf_size	int	30
p	int	2
metric	str or callable	minkowski
metric_params	dict	None
n_jobs	int	None

Table 31: The parameters and their default values from the sklearn algorithm for random forest.

Parameter	Type	Default Value
n_estimators	int	100
criterion	str	gini
max_depth	int	None
min_samples_split	int or float	2
min_samples_leaf	int or float	1
min_weight_fraction_leaf	float	0.0
max_features	str, int or float	sqrt
max_leaf_nodes	int	None
min_impurity_decrease	float	0.0
bootstrap	bool	True
oob_score	bool or callable	False
n_jobs	int	None
random_state	int, RandomState instance, or None	None
verbose	int	0
warm_start	bool	False
class_weight	str, dict, or list of dicts	None
ccp_alpha	non-negative float	0.0
max_samples	int or float	None

Table 32: The parameters and their default values from the sklearn algorithm for logistic regression.

Parameter	Type	Default Value
penalty	str	l2
dual	bool	False
tol	float	1e-4
C	float	1.0
fit_intercept	bool	True
intercept_scaling	float	1
class_weight	dict or balanced	None
random_state	int, RandomState instance	None
solver	str	lbfgs
max_iter	int	100
multi_class	str auto	
verbose	int	0
warm_start	bool	False
n_jobs	int	None
l1_ratio	float	None

Table 33: The parameters and their default values from the sklearn algorithm for support vector classifier.

Parameter	Type	Default Value
penalty	str	l2
loss	str	squared_hinge
dual	str or bool	True
tol	float	1e-4
C	float	1.0
multi_class	str	ovr
fit_intercept	bool	True
intercept_scaling	float	1.0
class_weight	dict or balanced	None
verbose	int	0
random_state	int, RandomState instance or None	None
max_iter	int	1000

Table 34: The parameters and their default values from the sklearn algorithm for isolation forest.

Parameter	Type	Default Value
n_estimators	int	100
max_samples	auto, int or float	auto
contamination	auto or float	auto
max_features	int or float	1.0
bootstrap	bool	False
n_jobs	int	None
random_state	int, RandomState instance or None	None
verbose	int	0
warm_start	bool	False

Table 35: The parameters and their default values from the sklearn algorithm for one-class support vector machine.

Parameter	Type	Default Value
kernel	linear, poly, rbf, sigmoid, precomputed, or callable	rbf
degree	int	3
gamma	scale, auto, or float	scale
coef0	float	0.0
tol	float	1e-3
nu	float	0.5
shrinking	bool	True
cache_size	float	200
verbose	bool	False
max_iter	int	-1

Table 36: The parameters and their default values from the sklearn algorithm for the local outlier factor.

Parameter	Type	Default Value
n_neighbors	int	20
algorithm	auto, ball_tree, kd_tree, or brute	auto
leaf_size	int	30
metric	str or callable	minkowski
p	int	2
metric_params	dict	None
contamination	auto or float	auto
novelty	bool	False
n_jobs	int	None

A.7 Supplemental Data

A.7.1 Sample Autoencoder Architecture Ablation

We present a number of figures representative of the ablation studies used to choose our autoencoder architecture. These initial ablation studies were carried out across five NID scenarios and used to determine autoencoder architecture.

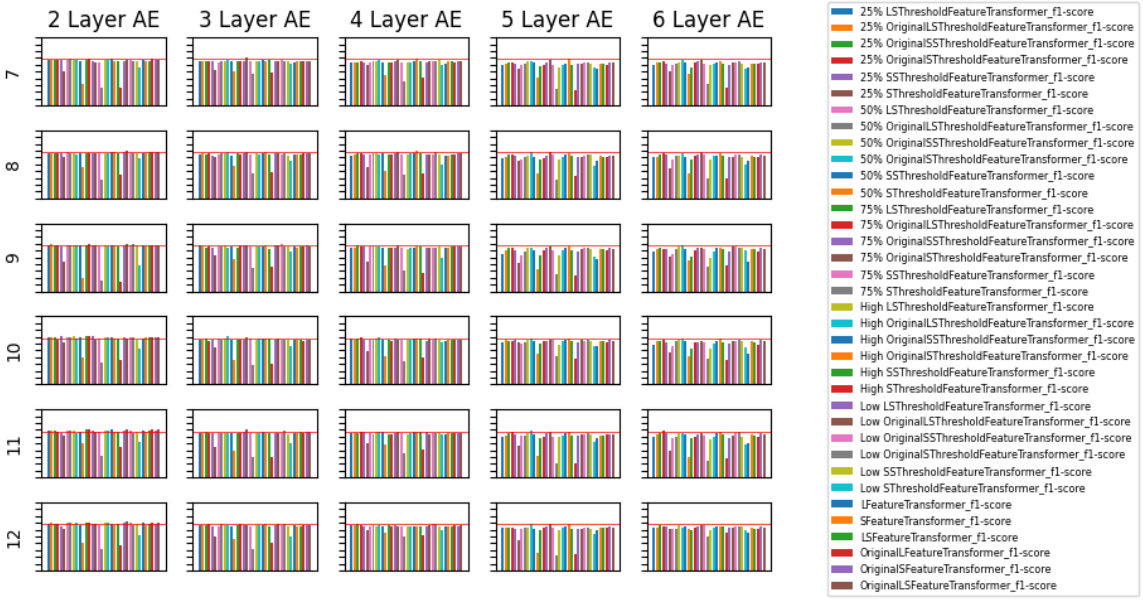


Figure 44: A sample of one of many ablation studies used to determine what autoencoder architecture to use for our research. Here we show an ablation study varying the number of layers in the encoder and decoder from two to six while also varying the bottleneck size from seven to twelve. While this figure is specific to the CTU-13 Scenario 5 dataset, it was used to determine a general autoencoder architecture to use in later research.

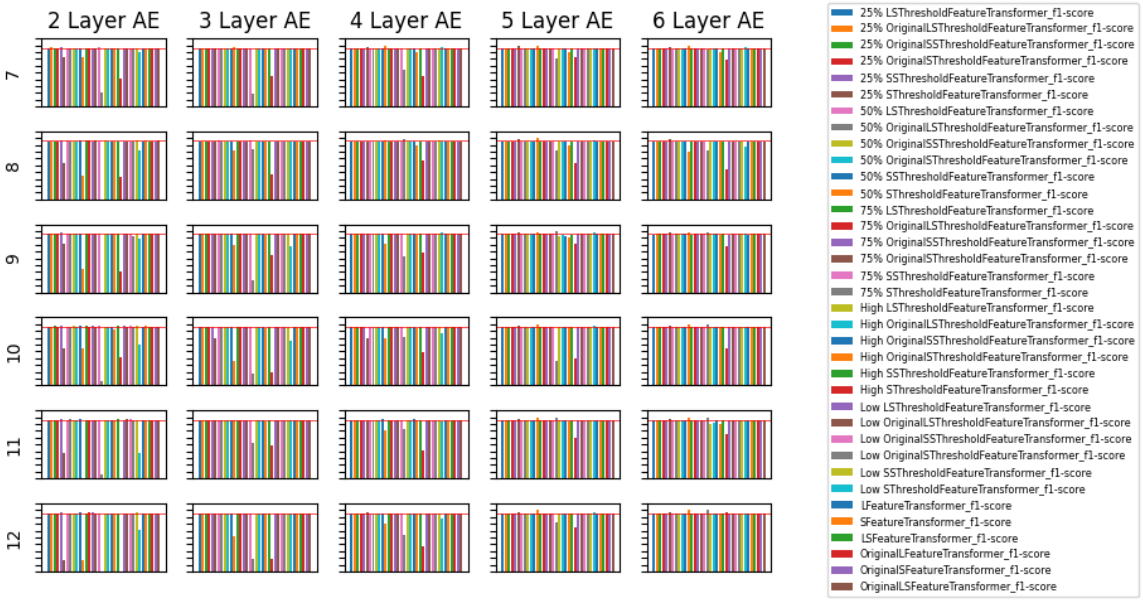


Figure 45: A sample of one of many ablation studies used to determine what autoencoder architecture to use for our research. Here we show an ablation study varying the number of layers in the encoder and decoder from two to six while also varying the bottleneck size from seven to twelve. While this figure is specific to the CTU-13 Scenario 6 dataset, it represents the general trend found across all datasets.

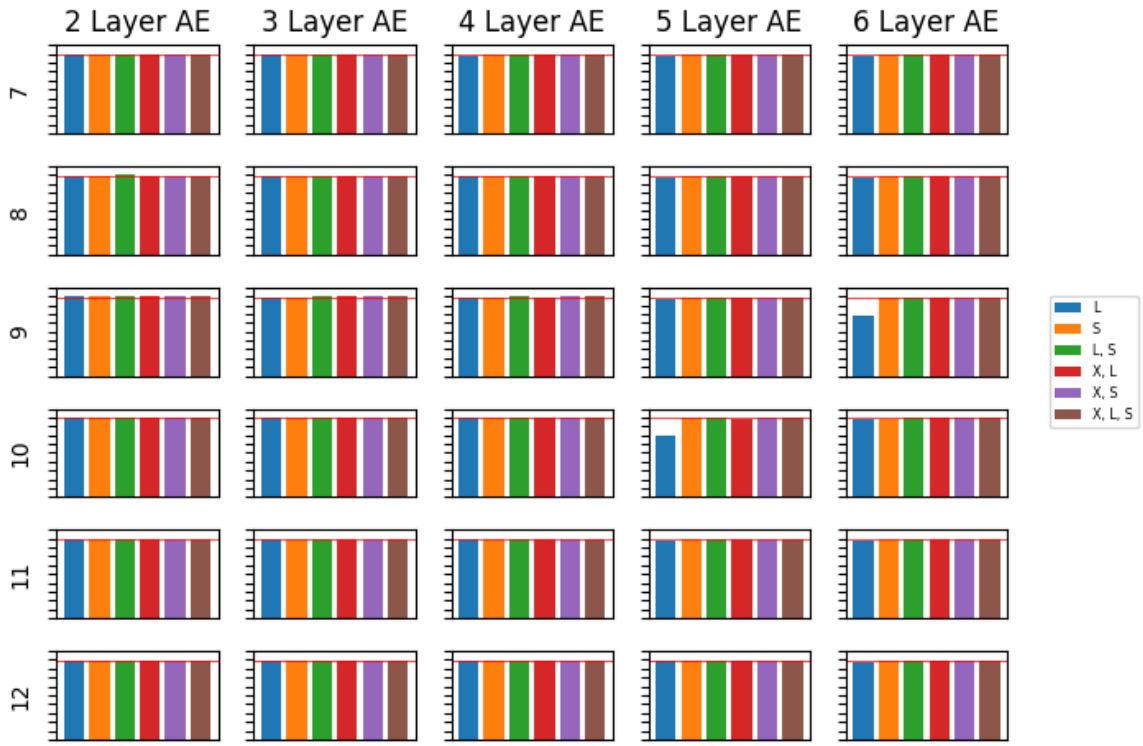


Figure 46: A sample of one of many ablation studies used to determine what autoencoder architecture to use for our research. Here we show an ablation study varying the number of layers in the encoder and decoder from two to six while also varying the bottleneck size from seven to twelve. While this figure is specific to the UNSW-NB15 dataset, it was used to determine a general autoencoder architecture to use in later research.

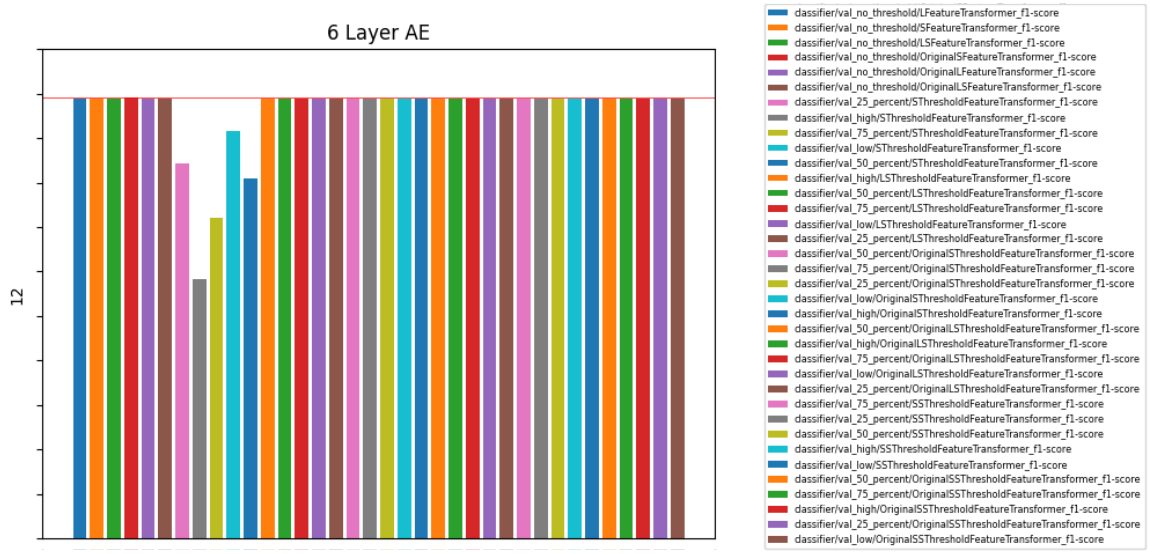


Figure 47: A sample of the extremes tested in our ablation studies used to determine what autoencoder architecture to use for our research. While this figure is specific to Tuesday's scenario from the CICIDS2017 dataset, it was used to determine a general autoencoder architecture to use in later research.

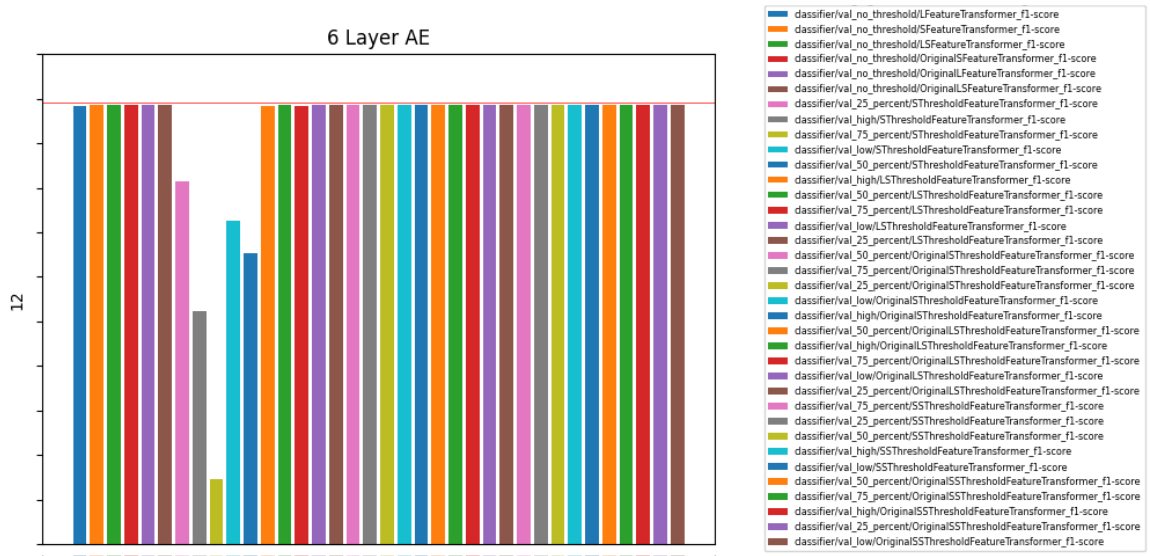


Figure 48: A sample of the extremes tested in our ablation studies used to determine what autoencoder architecture to use for our research. While this figure is specific to Wednesday's scenario from the CICIDS2017 dataset, it was used to determine a general autoencoder architecture to use in later research.

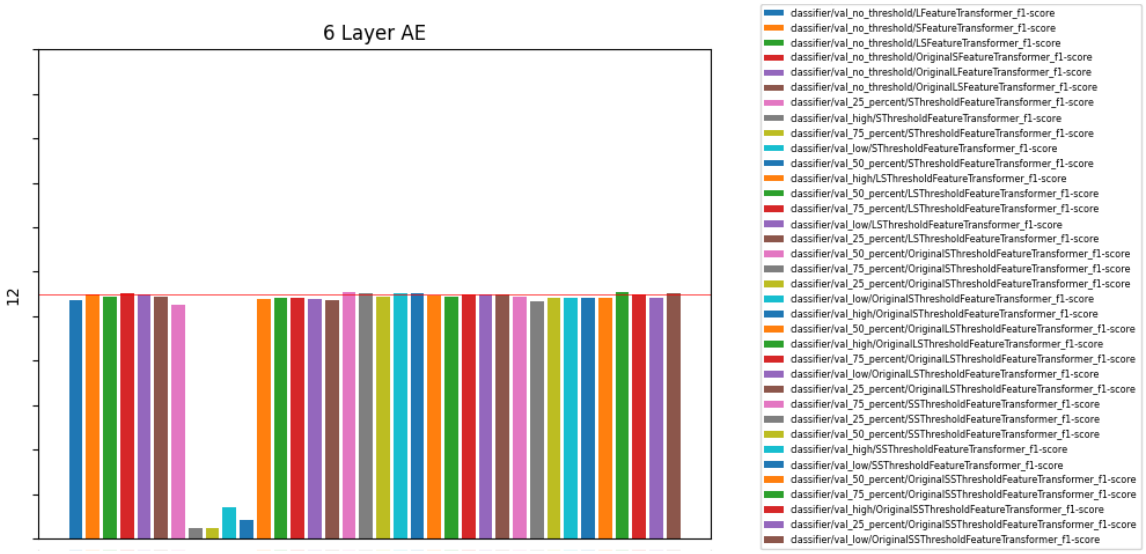


Figure 49: A sample of the extremes tested in our ablation studies used to determine what autoencoder architecture to use for our research. While this figure is specific to Thursday's scenario from the CICIDS2017 dataset, it was used to determine a general autoencoder architecture to use in later research.

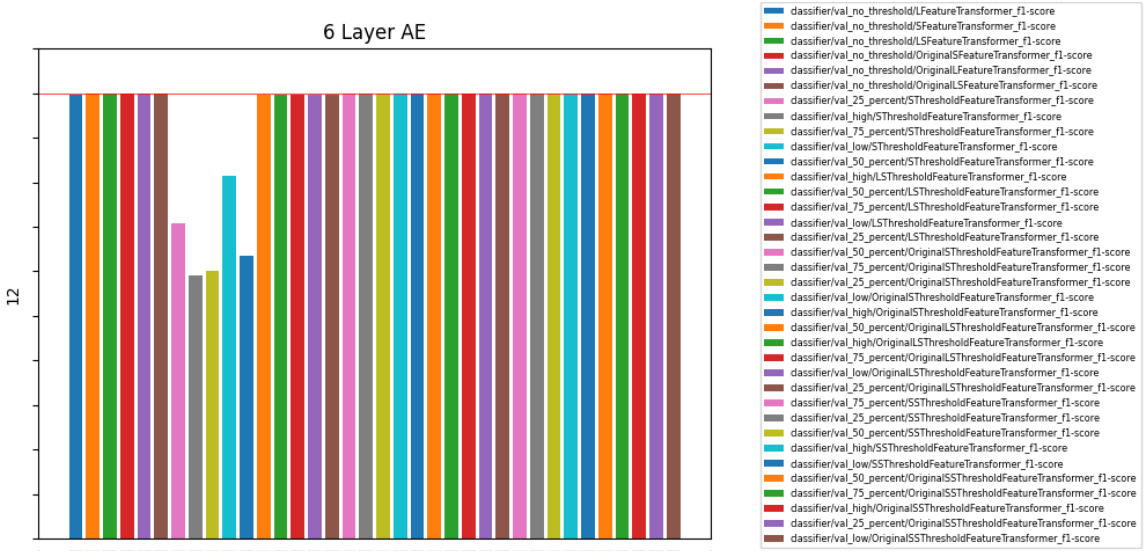


Figure 50: A sample of the extremes tested in our ablation studies used to determine what autoencoder architecture to use for our research. While this figure is specific to Friday's scenario from the CICIDS2017 dataset, it was used to determine a general autoencoder architecture to use in later research.

A.7.2 Precision and Recall Metrics

In this appendix we provide detailed information regarding the precision and recall metrics that support our f1-score reported measures. Tables 37, 38, 39, and 40 provide the precision and recall metrics for our experiments with MLP classifiers across the two encodings used in this dissertation. Tables 41, 42, 43, and 44 provide the precision and recall metrics for our experiments with RNN and LSTM classifiers using our simple encoding. Tables 45 and 46, along with Figures 51 and 52, provide the precision and recall metrics for our experiments using one-class classifiers.

Table 37: Mean precision results based on ten experiment executions using S in combination with X and L as input to our MLP classifier. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the precision scores were likely drawn from the same distribution. Values in blue indicate a precision and p-value that supports comparable performance compared to X. Values in green indicate a precision and p-value that supports we have improved performance compared to X. Values in orange indicate a precision and p-value that supports we have degraded performance compared to X. These values are included to supplement Table 11.

	NF-UNSW-NB15-V2		**NF-BoT-IoT-V2**		**NF-ToN-IoT-V2**		**NF-CSE-CIC-IDS2018-V2**	
	Precision	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**
X	0.881		0.995		0.963		0.997	
S	0.875	0.873	0.993	0.357	0.959	0.357	0.997	0.873
XS	0.885	0.873	0.994	0.873	0.968	0.873	0.997	0.873
LS	0.875	0.873	0.995	1.000	0.965	0.873	0.998	0.873
XLS	0.886	0.873	0.995	1.000	0.966	0.357	0.997	0.873

Table 38: Mean recall results based on ten experiment executions using S in combination with X and L as input to our MLP classifier. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the recall scores were likely drawn from the same distribution. Values in blue indicate a recall and p-value that supports comparable performance compared to X. Values in green indicate a recall and p-value that supports we have improved performance compared to X. Values in orange indicate a recall and p-value that supports we have degraded performance compared to X. These values are included to supplement Table 11.

	NF-UNSW-NB15-V2		**NF-BoT-IoT-V2**		**NF-ToN-IoT-V2**		**NF-CSE-CIC-IDS2018-V2**	
	Recall	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**
X	0.945		0.985		0.986		0.862	
S	0.931	0.873	0.986	0.357	0.979	0.357	0.861	0.873
XS	0.941	0.873	0.985	1.000	0.983	0.873	0.862	0.873
LS	0.975	0.357	0.985	0.873	0.985	0.873	0.861	0.873
XLS	0.940	0.873	0.985	1.000	0.986	1.000	0.862	0.873

Table 39: Mean precision test results based on ten experiment executions using S in combination with X and L as input to our MLP classifier using an alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the precision scores were likely drawn from the same distribution. Values in blue indicate a precision and p-value that supports comparable performance compared to X. Values in green indicate a precision and p-value that supports we have improved performance compared to X. These values are included to supplement Table 12.

	NF-UNSW-NB15-V2		**NF-BoT-IoT-V2**		**NF-ToN-IoT-V2**		**NF-CSE-CIC-IDS2018-V2**	
	Precision	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**
X	0.730		0.996		0.946		0.952	
S	0.765	0.079	0.997	0.357	0.950	0.357	0.981	0.008
XS	0.767	0.079	0.997	0.079	0.949	0.357	0.972	0.008
LS	0.755	0.357	0.997	0.008	0.948	0.873	0.975	0.008
XLS	0.776	0.079	0.997	0.079	0.950	0.873	0.973	0.008

Table 40: Mean recall test results based on ten experiment executions using S in combination with X and L as input to our MLP classifier using an alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the recall scores were likely drawn from the same distribution. Values in blue indicate a recall and p-value that supports comparable performance compared to X. Values in green indicate a recall and p-value that supports we have improved performance compared to X. These values are included to supplement Table 12.

	NF-UNSW-NB15-V2		NF-BoT-IoT-V2		NF-ToN-IoT-V2		NF-CSE-CIC-IDS2018-V2	
	Recall	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**
X	0.835		0.992		0.956		0.964	
S	0.849	0.873	0.992	1.000	0.957	0.873	0.969	0.079
XS	0.885	0.873	0.993	0.873	0.959	0.357	0.974	0.079
LS	0.919	0.008	0.992	1.000	0.961	0.079	0.971	0.079
XLS	0.868	0.873	0.993	0.873	0.956	0.873	0.975	0.008

Table 41: Mean precision test results based on ten experiment executions using S in combination with X and L as input to an RNN classifier using the alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the precision scores were likely drawn from the same distribution. Values in blue indicate a precision and p-value that supports comparable performance compared to X. Values in green indicate a precision and p-value that supports we have improved performance compared to X. Values in orange indicate a precision and p-value that supports we have degraded performance compared to X. This table supports the values reported in Table 14.

	UNSW-NB15		ToN-IoT		CTU13 Scenario 6		CTU13 Scenario 9		CTU13 Scenario 13	
	Precision	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**	**Precision**	**p-value**
X	0.979		0.872		0.897		0.814		0.692	
S	0.980	0.000	0.859	0.000	0.923	0.000	0.836	0.000	0.772	0.000
XS	0.981	0.000	0.913	0.000	0.899	0.000	0.830	0.000	0.790	0.000
LS	0.981	0.000	0.921	0.000	0.896	0.000	0.842	0.000	0.813	0.000
XLS	0.980	0.000	0.936	0.000	0.893	0.000	0.844	0.000	0.779	0.000

Table 42: Mean recall test results based on ten experiment executions using S in combination with X and L as input to an RNN classifier using the alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the recall scores were likely drawn from the same distribution. Values in blue indicate a recall and p-value that supports comparable performance compared to X. Values in green indicate a recall and p-value that supports we have improved performance compared to X. Values in orange indicate a recall and p-value that supports we have degraded performance compared to X. This table supports the values reported in Table 14.

	UNSW-NB15		ToN-IoT		CTU13 Scenario 6		CTU13 Scenario 9		CTU13 Scenario 13	
	Recall	p-value	Recall	p-value	Recall	p-value	Recall	p-value	Recall	p-value
X	0.981		0.981		0.765		0.821		0.778	
S	0.979	0.000	0.924	0.000	0.827	0.000	0.821	0.000	0.724	0.000
XS	0.979	0.000	0.959	0.000	0.778	0.000	0.836	0.000	0.815	0.000
LS	0.980	0.000	0.952	0.000	0.791	0.000	0.833	0.000	0.804	0.000
XLS	0.982	0.000	0.926	0.000	0.796	0.000	0.834	0.000	0.808	0.000

Table 43: Mean precision test results based on ten experiment executions using S in combination with X and L as input to an LSTM classifier using the alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the precision scores were likely drawn from the same distribution. Values in blue indicate a precision and p-value that supports comparable performance compared to X. Values in green indicate a precision and p-value that supports we have improved performance compared to X. Values in orange indicate a precision and p-value that supports we have degraded performance compared to X. This table supports the values reported in Table 15.

	UNSW-NB15		ToN-IoT		CTU13 Scenario 6		CTU13 Scenario 9		CTU13 Scenario 13	
	Precision	p-value	Precision	p-value	Precision	p-value	Precision	p-value	Precision	p-value
X	0.974		0.872		0.888		0.771		0.549	
S	0.975	0.000	0.819	0.000	0.871	0.000	0.795	0.000	0.626	0.000
XS	0.975	0.000	0.928	0.000	0.896	0.000	0.792	0.000	0.658	0.000
LS	0.975	0.000	0.939	0.000	0.887	0.000	0.792	0.000	0.649	0.000
XLS	0.976	0.000	0.919	0.000	0.891	0.000	0.798	0.000	0.633	0.000

Table 44: Mean recall test results based on ten experiment executions using S in combination with X and L as input to an LSTM classifier using the alternative simple encoding. P-values using the Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the recall scores were likely drawn from the same distribution. Values in blue indicate a recall and p-value that supports comparable performance compared to X. Values in green indicate a recall and p-value that supports we have improved performance compared to X. Values in orange indicate a recall and p-value that supports we have degraded performance compared to X. This table supports the values reported in Table 15.

	UNSW-NB15		**ToN-IoT**		**CTU13 Scenario 6**		**CTU13 Scenario 9**		**CTU13 Scenario 13**	
	Recall	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**	**Recall**	**p-value**
X	0.977		0.987		0.742		0.785		0.623	
S	0.975	0.000	0.854	0.000	0.770	0.000	0.798	0.000	0.655	0.000
XS	0.978	0.000	0.979	0.000	0.770	0.000	0.803	0.000	0.687	0.000
LS	0.977	0.000	0.907	0.000	0.767	0.000	0.801	0.000	0.682	0.000
XLS	0.977	0.000	0.987	0.000	0.787	0.000	0.799	0.000	0.681	0.000

Table 45: Baseline performance measures taken using default parameters for each one-class classifier. We report mean precision test results based on ten experiment executions using S in combination with X and L as input to our classifier. P-values using the two sample Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the precision scores were likely drawn from the same distribution. Values in blue indicate a precision and p-value that supports comparable performance compared to X. Values in green indicate a precision and p-value that supports we have improved performance compared to X. Values in orange indicate a precision and p-value that supports we have degraded performance compared to X. These results support the reported f1-scores in Tables 16 and 17.

	ToN-IoT		BoT-IoT		ToN-IoT (Simple)		IoT-23 Scenario 1		IoT-23 Scenario 13		IoT-23 Scenario 19		IoT-23 Scenario 20	
	Precision	p-value	Precision	p-value	Precision	p-value	Precision	p-value	Precision	p-value	Precision	p-value	Precision	p-value
Isolation Forest														
X	0.891	-	0.941	-	0.991	-	0.885	-	0.798	-	0.871	-	0.873	-
S	0.744	0.000	0.987	0.000	0.996	0.000	0.907	0.052	0.785	0.052	0.885	0.052	0.882	0.418
XS	0.832	0.002	0.975	0.000	0.996	0.000	0.900	0.168	0.782	0.052	0.889	0.787	0.883	0.052
LS	0.719	0.000	0.974	0.000	0.996	0.000	0.877	0.787	0.816	0.012	0.896	0.168	0.870	0.168
XLS	0.764	0.000	0.965	0.018	0.996	0.000	0.875	0.012	0.769	0.012	0.886	0.418	0.880	0.787
One-class Support Vector Machine														
X	0.754	-	0.852	-	0.978	-	0.796	-	0.652	-	0.819	-	0.757	-
S	0.692	0.000	0.884	0.000	0.980	0.000	0.788	0.000	0.649	0.000	0.832	0.002	0.824	0.000
XS	0.752	0.000	0.876	0.000	0.978	0.052	0.796	0.002	0.650	0.000	0.850	0.002	0.800	0.000
LS	0.760	0.000	0.875	0.000	0.978	0.012	0.797	0.000	0.653	0.000	0.880	0.000	0.775	0.168
XLS	0.738	0.115	0.870	0.000	0.978	0.000	0.797	0.000	0.653	0.000	0.874	0.002	0.771	0.168
Local Outlier Factor														
X	0.870	-	0.976	-	0.991	-	0.906	-	0.848	-	0.969	-	0.870	-
S	0.879	0.000	0.966	0.000	0.993	0.000	0.906	1.000	0.848	1.000	0.969	0.994	0.860	0.168
XS	0.871	0.168	0.975	0.000	0.991	0.002	0.905	0.994	0.848	1.000	0.969	0.994	0.869	0.052
LS	0.867	0.002	0.976	0.012	0.992	0.002	0.905	0.994	0.848	1.000	0.969	0.168	0.870	0.168
XLS	0.868	0.168	0.976	0.002	0.992	0.000	0.906	1.000	0.848	1.000	0.969	0.787	0.871	0.052

Table 46: Baseline performance measures taken using default parameters for each one-class classifier. We report mean recall test results based on ten experiment executions using S in combination with X and L as input to our classifier. P-values using the two sample Kolmogorov-Smirnov test are provided where values greater than 0.05 indicate the recall scores were likely drawn from the same distribution. Values in blue indicate a recall and p-value that supports comparable performance compared to X. Values in green indicate a recall and p-value that supports we have improved performance compared to X. Values in orange indicate a recall and p-value that supports we have degraded performance compared to X. These results support the reported f1-scores in Tables 16 and 17.

	ToN-IoT		BoT-IoT		ToN-IoT (Simple)		IoT-23 Scenario 1		IoT-23 Scenario 13		IoT-23 Scenario 19		IoT-23 Scenario 20	
	Recall	p-value	Recall	p-value	Recall	p-value	Recall	p-value	Recall	p-value	Recall	p-value	Recall	p-value
Isolation Forest														
X	0.183	-	0.252	-	0.905	-	1.000	-	1.000	-	0.462	-	0.556	-
S	0.033	0.000	0.637	0.000	0.612	0.000	0.998	0.168	1.000	1.000	0.4657	0.168	0.496	0.052
XS	0.065	0.000	0.366	0.052	0.866	0.012	1.0000	1.000	1.000	1.000	0.4835	0.168	0.511	0.012
LS	0.060	0.000	0.461	0.000	0.790	0.000	0.986	0.994	1.000	1.000	0.6057	0.418	0.554	0.168
XLS	0.060	0.000	0.365	0.230	0.878	0.002	0.992	1.000	1.000	1.000	0.5090	0.168	0.541	0.418
One-class Support Vector Machine														
X	0.505	-	0.741	-	0.900	-	1.000	-	1.000	-	0.509	-	0.588	-
S	0.374	0.000	0.988	0.000	0.994	0.000	1.000	1.000	1.000	1.000	0.6050	0.012	0.872	0.000
XS	0.496	0.000	0.911	0.000	0.901	0.000	1.000	1.000	1.000	1.000	0.7019	0.000	0.757	0.000
LS	0.519	0.000	0.905	0.000	0.901	0.012	1.000	1.000	1.000	1.000	0.8758	0.002	0.661	0.168
XLS	0.507	0.115	0.863	0.000	0.901	0.012	1.000	1.000	1.000	1.000	0.8452	0.000	0.654	0.168
Local Outlier Factor														
X	0.231	-	0.771	-	0.614	-	1.000	-	1.000	-	0.827	-	0.640	-
S	0.264	0.000	0.822	0.000	0.888	0.000	1.000	1.000	1.000	1.000	0.837	0.787	0.599	0.168
XS	0.233	0.168	0.802	0.000	0.630	0.002	1.000	1.000	1.000	1.000	0.831	0.994	0.640	1.000
LS	0.228	0.052	0.800	0.000	0.620	0.002	1.000	1.000	1.000	1.000	0.835	0.168	0.641	0.418
XLS	0.228	0.052	0.806	0.000	0.620	0.052	1.000	1.000	1.000	1.000	0.839	0.418	0.640	1.000

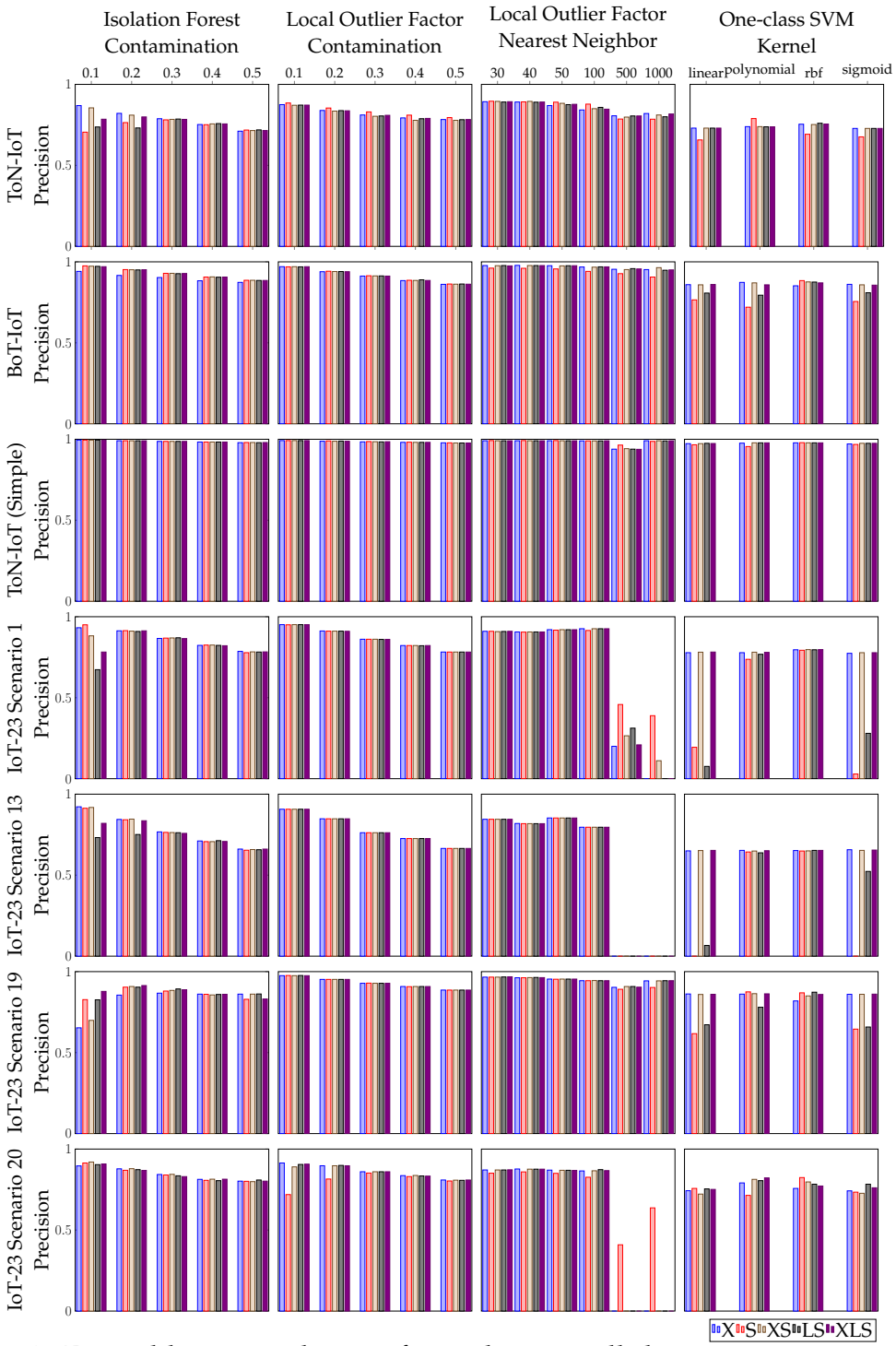


Figure 51: Four ablation studies performed across all datasets to compare using feature combinations that include S to just using X when the one-class classification algorithms have been optimized for performance. In general, these ablation studies show that our novel feature combinations generally outperform X or meet the same performance of X even when the classifiers are optimized beyond their baseline settings. This figure supports the f1-score reported in Figure 34.

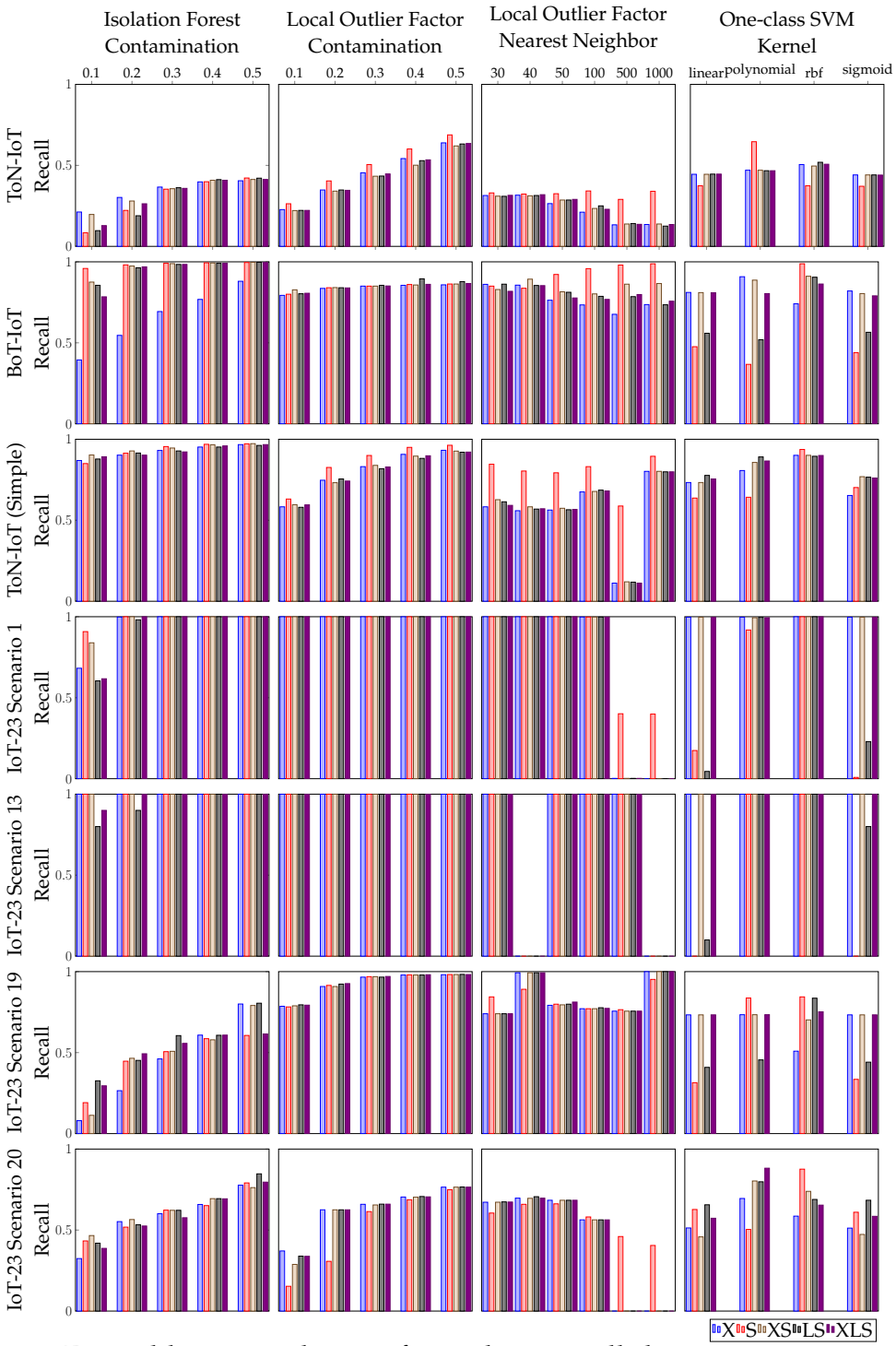


Figure 52: Four ablation studies performed across all datasets to compare using feature combinations that include S to just using X when the one-class classification algorithms have been optimized for performance. In general, these ablation studies show that our novel feature combinations generally outperform X or meet the same performance of X even when the classifiers are optimized beyond their baseline settings. This figure supports the f1-score reported in Figure 34.

References

[1] Kdd cup 99. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html, 1999. Accessed: 2022-08-08.

[2] ACOSTA, J. C., MEDINA, S., ELLIS, J., CLARKE, L., RIVAS, V., AND NEWCOMB, A. Network data curation toolkit: Cybersecurity data collection, aided-labeling, and rule generation. In *MILCOM 2021 - 2021 IEEE Military Communications Conference (MILCOM)* (2021), pp. 849–854.

[3] AGUILGOST, F., SIMMEZQUITA, E., MARNTORDERA, E., AND HUSSAIN, A. A machine learning ids for known and unknown anomalies. In *2022 18th International Conference on the Design of Reliable Communication Networks (DRCN)* (2022), pp. 1–5.

[4] AHMED, M., MAHMOOD, A. N., AND HU, J. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications 60* (Jan. 2016), 19–31.

[5] AKGUN, D., HIZAL, S., AND CAVUSOGLU, U. A new DDoS attacks intrusion detection model based on deep learning for cybersecurity. *Computers and Security 118* (July 2022), 102748.

[6] ALMOMANI, I., AL-KASASBEH, B., AND AL-AKHRAS, M. WSN-DS: A dataset for intrusion detection systems in wireless sensor networks. *Journal of Sensors 2016* (2016), 1–16.

[7] AN, J., AND CHO, S. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE 2*, 1 (2015), 1–18.

[8] ANDRESINI, G., APPICE, A., MAURO, N. D., LOGLISCI, C., AND MALERBA, D. Exploiting the auto-encoder residual error for intrusion detection. In *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS & PW)* (June 2019), IEEE.

[9] APRUZZESE, G., ANDREOLINI, M., COLAJANNI, M., AND MARCHETTI, M. Hardening random forest cyber detectors against adversarial attacks. *IEEE Transactions on Emerging Topics in Computational Intelligence 4*, 4 (2020), 427–439.

[10] APRUZZESE, G., LASKOV, P., DE OCA, E. M., MALLOULI, W., RAPA, L. B., GRAMMATOPOULOS, A. V., AND FRANCO, F. D. The role of machine learning in cybersecurity. *Digital Threats: Research and Practice* (July 2022).

[11] AYGUN, R. C., AND YAVUZ, A. G. Network anomaly detection with stochastically improved autoencoder based models. In *2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud)* (June 2017), IEEE.

[12] BAGUI, S., AND LI, K. Resampling imbalanced data for network intrusion detection datasets. *Journal of Big Data 8*, 1 (Jan. 2021).

[13] BAHADUR, N., LEWANDOWSKI, B., AND PAFFENROTH, R. Dimension estimation using autoencoders and application. In *Deep Learning Applications, Volume 3*. Springer, 2022, pp. 95–121.

[14] BASATI, A., AND FAGHIH, M. M. PDAE: Efficient network intrusion detection in IoT using parallel deep auto-encoders. *Information Sciences 598* (June 2022), 57–74.

[15] BHATTACHARYA, S., AND SELVAKUMAR, S. Ssenet-2014 dataset: A dataset for detection of multiconnection attacks. In *Proceedings of the 2014 3rd International Conference on Eco-Friendly Computing and Communication Systems* (USA, 2014), ICECCS '14, IEEE Computer Society, p. 121126.

[16] BRAUCKHOFF, D., WAGNER, A., AND MAY, M. Flame: A flow-level anomaly modeling engine. In *CSET* (2008).

[17] BREUNIG, M. M., KRIEGEL, H.-P., NG, R. T., AND SANDER, J. Lof: Identifying density-based local outliers. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2000), SIGMOD '00, Association for Computing Machinery, p. 93104.

[18] CAO, V. L., NICOLAU, M., AND MCDERMOTT, J. A hybrid autoencoder and density estimation model for anomaly detection. In *Parallel Problem Solving from Nature – PPSN XIV* (Cham, 2016), J. Handl, E. Hart, P. R. Lewis, M. López-Ibáñez, G. Ochoa, and B. Paechter, Eds., Springer International Publishing, pp. 717–726.

[19] CERMAK, M., JIRSIK, T., VELAN, P., KOMARKOVA, J., SPACEK, S., DRASAR, M., AND PLESNIK, T. Towards provable network traffic measurement and analysis via semi-labeled trace datasets. In *2018 Network Traffic Measurement and Analysis Conference (TMA)* (2018), pp. 1–8.

[20] CHALAPATHY, R., AND CHAWLA, S. Deep learning for anomaly detection: A survey. *arXiv preprint arXiv:1901.03407* (2019).

[21] CHALAPATHY, R., MENON, A. K., AND CHAWLA, S. Anomaly detection using one-class neural networks, 2018.

[22] CHANDOLA, V., BANERJEE, A., AND KUMAR, V. Anomaly detection: A survey. *ACM Comput. Surv. 41*, 3 (jul 2009).

[23] CHEN, Z., YEO, C. K., LEE, B. S., AND LAU, C. T. Autoencoder-based network anomaly detection. In *2018 Wireless Telecommunications Symposium (WTS)* (2018), pp. 1–5.

[24] CHOU, D., AND JIANG, M. A survey on data-driven network intrusion detection. *ACM Computing Surveys 54*, 9 (Dec. 2022), 1–36.

[25] CLAISE, B. Specification of the ip flow information export (ipfix) protocol for the exchange of ip traffic flow information, 2008.

[26] CORDERO, C. G., VASILOMANOLAKIS, E., MILANOV, N., KOCH, C., HAUSHEER, D., AND MHLHUSER, M. Id2t: A diy dataset creation toolkit for intrusion detection systems. In *2015 IEEE Conference on Communications and Network Security (CNS)* (2015), pp. 739–740.

[27] CORDERO, C. G., VASILOMANOLAKIS, E., WAINAKH, A., MÜHLHÄUSER, M., AND NADJM-TEHRANI, S. On generating network traffic datasets with synthetic attacks for intrusion detection. *ACM Trans. Priv. Secur. 24*, 2 (jan 2021).

[28] DASGUPTA, D., AKHTAR, Z., AND SEN, S. Machine learning in cybersecurity: a comprehensive survey. *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology 19*, 1 (Sept. 2020), 57–106.

[29] DE CARVALHO BERTOLI, G., JUNIOR, L. A. P., SAOTOME, O., AND DOS SANTOS, A. L. Generalizing intrusion detection for heterogeneous networks: A stacked-unsupervised federated learning approach. *Computers and Security 127* (Apr. 2023), 103106.

[30] ELSAYED, M. S., LE-KHAC, N.-A., DEV, S., AND JURCUT, A. D. Ddosnet: A deep-learning model for detecting network attacks. In *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)* (2020), pp. 391–396.

[31] ENGELEN, G., RIMMER, V., AND JOOSEN, W. Troubleshooting an intrusion detection dataset: the cicids2017 case study. In *2021 IEEE Security and Privacy Workshops (SPW)* (2021), pp. 7–12.

[32] FERRIYAN, A., THAMRIN, A. H., TAKEDA, K., AND MURAI, J. Generating network intrusion detection dataset based on real and encrypted synthetic attack traffic. *Applied Sciences 11*, 17 (2021).

[33] FU, Y., DU, Y., CAO, Z., LI, Q., AND XIANG, W. A deep learning model for network intrusion detection with imbalanced data. *Electronics 11*, 6 (Mar. 2022), 898.

[34] GARCÍA, S., GRILL, M., STIBOREK, J., AND ZUNINO, A. An empirical comparison of botnet detection methods. *Computers and Security 45* (Sept. 2014), 100–123.

[35] GARCIA, S., PARMISANO, A., AND ERQUIAGA, M. J. Iot-23: A labeled dataset with malicious and benign iot network traffic, 2020.

[36] GHAFOORI, Z., ERFANI, S. M., RAJASEGARAR, S., BEZDEK, J. C., KARUNASEKERA, S., AND LECKIE, C. Efficient unsupervised parameter estimation for one-class support vector machines. *IEEE Transactions on Neural Networks and Learning Systems 29*, 10 (2018), 5057–5070.

[37] GHANEM, K., APARICIO-NAVARRO, F. J., KYRIAKOPOULOS, K. G., LAMBOTHARAN, S., AND CHAMBERS, J. A. Support vector machine for network intrusion and cyber-attack detection. In *2017 sensor signal processing for defence conference (SSPD)* (2017), IEEE, pp. 1–5.

[38] GONZALEZ-GRANADILLO, G., BEDOYA, A., AND DIAZ, R. An improved live anomaly detection system (i-lads) based on deep learning algorithm. In *Proceedings of the 18th SECRYPT Conference, Online, Streaming* (2021), pp. 6–8.

[39] GOODFELLOW, I., BENGIO, Y., AND COURVILLE, A. *Deep learning*. MIT press, 2016.

[40] GOTTWALT, F., CHANG, E., AND DILLON, T. CorrCorr: A feature selection method for multivariate correlation network anomaly detection techniques. *Computers and Security 83* (June 2019), 234–245.

[41] HAN, D., WANG, Z., CHEN, W., ZHONG, Y., WANG, S., ZHANG, H., YANG, J., SHI, X., AND YIN, X. Deepaid: Interpreting and improving deep learning-based anomaly detection in security applications. *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Nov 2021).

[42] HE, H., AND GARCIA, E. A. Learning from imbalanced data. *IEEE Transactions on Knowledge and Data Engineering 21*, 9 (2009), 1263–1284.

[43] HERSHEY, Q. *Exploring Neural Network Structure through Iterative Neural Networks: Connections to Dynamical Systems*. PhD thesis, Worcester Polytechnic Institute, 2023.

[44] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural Computation 9*, 8 (1997), 1735–1780.

[45] HODGES, J. L. The significance probability of the smirnov two-sample test. *Arkiv för Matematik 3*, 5 (1958), 469–486.

[46] JAMES, G., WITTEN, D., HASTIE, T., AND TIBSHIRANI, R. *An introduction to statistical learning*, vol. 112. Springer, 2013.

[47] JIN, Y., SHEN, Y., ZHANG, G., AND ZHI, H. The model of network security situation assessment based on random forest. In *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)* (2016), pp. 977–980.

[48] KAAN SARICA, A., AND ANGIN, P. A novel sdn dataset for intrusion detection in iot networks. In *2020 16th International Conference on Network and Service Management (CNSM)* (2020), pp. 1–5.

[49] KENYON, A., DEKA, L., AND ELIZONDO, D. Are public intrusion datasets fit for purpose characterising the state of the art in intrusion event datasets. *Computers & Security 99* (2020), 102022.

[50] KHAN, M. A. HCRNNIDS: Hybrid convolutional recurrent neural network-based network intrusion detection system. *Processes 9*, 5 (May 2021), 834.

[51] KIM, J., SHIN, N., JO, S. Y., AND KIM, S. H. Method of intrusion detection using deep neural network. In *2017 IEEE international conference on big data and smart computing (BigComp)* (2017), IEEE, pp. 313–316.

[52] KOMISAREK, M., PAWLICKI, M., KOZIK, R., HOUBOWICZ, W., AND CHORA, M. How to effectively collect and process network data for intrusion detection? *Entropy 23*, 11 (2021).

[53] KORONIOTIS, N., MOUSTAFA, N., SITNIKOVA, E., AND TURNBULL, B. Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-IoT dataset. *Future Generation Computer Systems 100* (Nov. 2019), 779–796.

[54] KRAMER, M. A. Nonlinear principal component analysis using autoassociative neural networks. *AIChE journal 37*, 2 (1991), 233–243.

[55] KUMAR, S., AND CARLEY, K. M. Approaches to understanding the motivations behind cyber attacks. In *2016 IEEE Conference on Intelligence and Security Informatics (ISI)* (2016), pp. 307–309.

[56] LAHASAN, B., AND SAMMA, H. Optimized deep autoencoder model for internet of things intruder detection. *IEEE Access 10* (2022), 8434–8448.

[57] LALLIE, H. S., SHEPHERD, L. A., NURSE, J. R., EROLA, A., EPIPHANIOU, G., MAPLE, C., AND BELLEKENS, X. Cyber security in the age of COVID-19: A timeline and analysis of cyber-crime and cyber-attacks during the pandemic. *Computers & Security 105* (June 2021), 102248.

[58] LANVIN, M., GIMENEZ, P.-F., HAN, Y., MAJORCZYK, F., MÉ, L., AND TOTEL, E. Errors in the CICIDS2017 dataset and the significant differences in detection performances it makes. In *CRiSIS 2022 - International Conference on Risks and Security of Internet and Systems* (Sousse, Tunisia, Dec. 2022), pp. 1–16.

[59] LAVINIA, Y., DURAIRAJAN, R., REJAIE, R., AND WILLINGER, W. Challenges in using ml for networking research: How to label if you must. In *Proceedings of the Workshop on Network Meets AI & ML* (New York, NY, USA, 2020), NetAI '20, Association for Computing Machinery, p. 2127.

[60] LAYEGHY, S., GALLAGHER, M., AND PORTMANN, M. Benchmarking the benchmark – analysis of synthetic nids datasets, 2021.

[61] LIU, F. T., TING, K. M., AND ZHOU, Z.-H. Isolation-based anomaly detection. *ACM Trans. Knowl. Discov. Data 6*, 1 (mar 2012).

[62] LONG, C., XIAO, J., WEI, J., ZHAO, J., WAN, W., AND DU, G. Autoencoder ensembles for network intrusion detection. In *2022 24th International Conference on Advanced Communication Technology (ICACT)* (Feb. 2022), IEEE.

[63] LOPES, I. O., ZOU, D., ABDULQADDER, I. H., RUAMBO, F. A., YUAN, B., AND JIN, H. Effective network intrusion detection via representation learning: A denoising AutoEncoder approach. *Computer Communications 194* (Oct. 2022), 55–65.

[64] MACAS, M., WU, C., AND FUERTES, W. A survey on deep learning for cybersecurity: Progress, challenges, and opportunities. *Computer Networks 212* (July 2022), 109032.

[65] MACIÁ-FERNÁNDEZ, G., CAMACHO, J., MAGÁN-CARRIÓN, R., GARCÍA-TEODORO, P., AND THERÓN, R. UGR'16: A new dataset for the evaluation of cyclostationarity-based network IDSs. *Computers and Security 73* (Mar. 2018), 411–424.

[66] MAGAN-CARRION, R., URDA, D., DIAZ-CANO, I., AND DORRONSORO, B. Improving the reliability of network intrusion detection systems through

dataset aggregation. *IEEE Transactions on Emerging Topics in Computing* (2022), 1–1.

[67] MBONA, I., AND ELOFF, J. H. P. Detecting zero-day intrusion attacks using semi-supervised machine learning approaches. *IEEE Access 10* (2022), 69822–69838.

[68] MEIDAN, Y., BOHADANA, M., MATHOV, Y., MIRSKY, Y., SHABTAI, A., BREITENBACHER, D., AND ELOVICI, Y. N-baiotnetwork-based detection of iot botnet attacks using deep autoencoders. *IEEE Pervasive Computing 17*, 3 (2018), 12–22.

[69] MIRSKY, Y., DOITSHMAN, T., ELOVICI, Y., AND SHABTAI, A. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089* (2018).

[70] MOUSTAFA, N. A new distributed architecture for evaluating AI-based security systems at the edge: Network TON_IoT datasets. *Sustainable Cities and Society 72* (Sept. 2021), 102994.

[71] MOUSTAFA, N., CREECH, G., AND SLAY, J. *Big Data Analytics for Intrusion Detection System: Statistical Decision-Making Using Finite Dirichlet Mixture Models*. Springer International Publishing, Cham, 2017, pp. 127–156.

[72] MOUSTAFA, N., HU, J., AND SLAY, J. A holistic review of network anomaly detection systems: A comprehensive survey. *Journal of Network and Computer Applications 128* (Feb. 2019), 33–55.

[73] MOUSTAFA, N., AND SLAY, J. Unsw-nb15: a comprehensive data set for network intrusion detection systems (unsw-nb15 network data set). In *2015 Military Communications and Information Systems Conference (MilCIS)* (2015), pp. 1–6.

[74] MUSHTAQ, E., ZAMEER, A., UMER, M., AND ABBASI, A. A. A two-stage intrusion detection system with auto-encoder and LSTMs. *Applied Soft Computing 121* (May 2022), 108768.

[75] NANDA, N. B., AND PARIKH, A. Hybrid approach for network intrusion detection system using random forest classifier and rough set theory for rules generation. In *Communications in Computer and Information Science*. Springer Singapore, 2019, pp. 274–287.

[76] ORTEGA-FERNANDEZ, I., SESTELO, M., BURGUILLO, J. C., AND PIÑÓN-BLANCO, C. Network intrusion detection system for DDoS attacks in ICS using deep autoencoders. *Wireless Networks* (Jan. 2023).

[77] PAFFENROTH, R., DU TOIT, P., NONG, R., SCHARF, L., JAYASUMANA, A. P., AND BANDARA, V. Space-time signal processing for distributed pattern detection in sensor networks. *IEEE Journal of Selected Topics in Signal Processing 7*, 1 (2013), 38–49.

[78] PAFFENROTH, R. C., AND ZHOU, C. Modern machine learning for cyber-defense and distributed denial-of-service attacks. *IEEE Engineering Management Review 47*, 4 (2019), 80–85.

[79] PANG, G., SHEN, C., CAO, L., AND HENGEL, A. V. D. Deep learning for anomaly detection: A review. *ACM Comput. Surv. 54*, 2 (mar 2021).

[80] RAJASINGHE, N., SAMARABANDU, J., AND WANG, X. Insecs-dcs: A highly customizable network intrusion dataset creation framework. In *2018 IEEE Canadian Conference on Electrical & Computer Engineering (CCECE)* (2018), pp. 1–4.

[81] RAVI, V., CHAGANTI, R., AND ALAZAB, M. Recurrent deep learning-based feature fusion ensemble meta-classifier approach for intelligent network intrusion detection system. *Computers and Electrical Engineering 102* (Sept. 2022), 108156.

[82] RING, M., WUNDERLICH, S., GRÜDL, D., LANDES, D., AND HOTHO, A. Creation of flow-based data sets for intrusion detection. *Journal of Information Warfare 16*, 4 (2017), 41–54.

[83] RING, M., WUNDERLICH, S., GRÜDL, D., LANDES, D., AND HOTHO, A. Flow-based benchmark data sets for intrusion detection. In *Proceedings of the 16th European conference on cyber warfare and security* (2017).

[84] RING, M., WUNDERLICH, S., SCHEURING, D., LANDES, D., AND HOTHO, A. A survey of network-based intrusion detection data sets. *Computers and Security 86* (Sept. 2019), 147–167.

[85] RUMELHART, D. E., HINTON, G. E., AND WILLIAMS, R. J. Learning internal representations by error propagation. Tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[86] SABAR, N. R., YI, X., AND SONG, A. A bi-objective hyper-heuristic support vector machines for big data cyber-security. *IEEE Access 6* (2018), 10421–10431.

[87] SAK, H., SENIOR, A., AND BEAUFAYS, F. Long short-term memory based recurrent neural network architectures for large vocabulary speech recognition, 2014.

[88] SARHAN, M., LAYEGHY, S., MOUSTAFA, N., AND PORTMANN, M. Netflow datasets for machine learning-based network intrusion detection systems. In *Big Data Technologies and Applications*. Springer, 2020, pp. 117–135.

[89] SARHAN, M., LAYEGHY, S., MOUSTAFA, N., AND PORTMANN, M. A cyber threat intelligence sharing scheme based on federated learning for network intrusion detection, 2021.

[90] SARHAN, M., LAYEGHY, S., AND PORTMANN, M. Evaluating standard feature sets towards increased generalisability and explainability of ml-based network intrusion detection, 2021.

[91] SARHAN, M., LAYEGHY, S., AND PORTMANN, M. Towards a standard feature set for network intrusion detection system datasets. *Mobile Networks and Applications 27*, 1 (Nov. 2021), 357–370.

[92] SCHÖLKOPF, B., WILLIAMSON, R. C., SMOLA, A., SHAWE-TAYLOR, J., AND PLATT, J. Support vector method for novelty detection. In *Advances in Neural Information Processing Systems* (1999), S. Solla, T. Leen, and K. Müller, Eds., vol. 12, MIT Press.

[93] SHARAFALDIN, I., GHARIB, A., LASHKARI, A. H., AND GHORBANI, A. A. Towards a reliable intrusion detection benchmark dataset. *Software Networking 2018*, 1 (2018), 177–200.

[94] SHARAFALDIN, I., LASHKARI, A. H., AND GHORBANI, A. A. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp 1* (2018), 108–116.

[95] SHARAFALDIN, I., LASHKARI, A. H., AND GHORBANI, A. A. Toward generating a new intrusion detection dataset and intrusion traffic characterization. In *International Conference on Information Systems Security and Privacy* (2018).

[96] SHARIF, M. H. U., AND MOHAMMED, M. A. A literature review of financial losses statistics for cyber security and future trend. *World Journal of Advanced Research and Reviews 15*, 1 (2022), 138–156.

[97] SOMMER, R., AND PAXSON, V. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy* (2010), pp. 305–316.

[98] SONG, Y., HYUN, S., AND CHEONG, Y.-G. Analysis of autoencoders for network intrusion detection. *Sensors 21*, 13 (June 2021), 4294.

[99] SYSTEMS, C. Cisco ios netflow version 9 flow record format, 2011.

[100] TANG, T. A., MHAMDI, L., MCLERNON, D., ZAIDI, S. A. R., AND GHOGHO, M. Deep learning approach for network intrusion detection in software defined networking. In *2016 International Conference on Wireless Networks and Mobile Communications (WINCOM)* (2016), pp. 258–263.

[101] TAVALLAEE, M., BAGHERI, E., LU, W., AND GHORBANI, A. A. A detailed analysis of the kdd cup 99 data set. In *2009 IEEE Symposium on Computational Intelligence for Security and Defense Applications* (2009), Ieee, pp. 1–6.

[102] TSVETANOV, T., AND SLARIA, S. The effect of the colonial pipeline shutdown on gasoline prices. *Economics Letters 209* (Dec. 2021), 110122.

[103] ULLAH, I., AND MAHMOUD, Q. H. Design and development of rnn anomaly detection model for iot networks. *IEEE Access 10* (2022), 62722–62750.

[104] WANG, W., JIAN, S., TAN, Y., WU, Q., AND HUANG, C. Representation learning-based network intrusion detection system by capturing explicit and implicit feature interactions. *Computers & Security 112* (Jan. 2022), 102537.

[105] WIDMER, G., AND KUBAT, M. Learning in the presence of concept drift and hidden contexts. *Machine Learning 23*, 1 (1996), 69–101.

[106] WOLSING, K., WAGNER, E., SAILLARD, A., AND HENZE, M. Ipal: Breaking up silos of protocol-dependent and domain-specific industrial intrusion detection systems, 2021.

[107] XU, W., AND FAN, Y. Intrusion detection systems based on logarithmic autoencoder and XGBoost. *Security and Communication Networks 2022* (Apr. 2022), 1–8.

[108] YANG, Z., LIU, X., LI, T., WU, D., WANG, J., ZHAO, Y., AND HAN, H. A systematic literature review of methods and datasets for anomaly-based network intrusion detection. *Computers & Security* (2022), 102675.

[109] YEHEZKEL, A., ELYASHIV, E., AND SOFFER, O. Network anomaly detection using transfer learning based on auto-encoders loss normalization. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security* (2021), pp. 61–71.

[110] YOUSEFI-AZAR, M., VARADHARAJAN, V., HAMEY, L., AND TUPAKULA, U. Autoencoder-based feature learning for cyber security applications. In *2017 International Joint Conference on Neural Networks (IJCNN)* (2017), pp. 3854–3861.

[111] ZAVRAK, S., AND SKEFIYELI, M. Anomaly-based intrusion detection from network flow features using variational autoencoder. *IEEE Access 8* (2020), 108346–108358.

[112] ZHANG, H., GE, L., ZHANG, G., FAN, J., LI, D., AND XU, C. A two-stage intrusion detection method based on light gradient boosting machine and autoencoder. *Mathematical Biosciences and Engineering 20*, 4 (2023), 6966–6992.

[113] ZHANG, T., CHEN, W., LIU, Y., AND WU, L. An intrusion detection method based on stacked sparse autoencoder and improved gaussian mixture model. *Computers and Security* (Feb. 2023), 103144.

[114] ZHOU, C., AND PAFFENROTH, R. C. Anomaly detection with robust deep autoencoders. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining* (New York, NY, USA, Aug. 2017), KDD '17, Association for Computing Machinery, pp. 665–674.