MACHINE LEARNING FOR NETWORK TRAFFIC CLASSIFICATION UNDER LABELED DATA AND TRAINING TIME CONSTRAINTS

by

Jason P. Hussey

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Doctor of Philosophy (Computer Science).

Golden, Colorado

Date _____

Signed: _____

Jason P. Hussey

Signed: _____

Dr. Tracy K. Camp
Thesis Advisor

Signed: _____

Dr. Kerri A. Stone
Thesis Advisor

Golden, Colorado

Date _____

Signed: _____

Dr. Iris Bahar
Department Head and Professor
Department of Computer Science

ABSTRACT

This thesis investigates using machine learning (including deep learning) for network traffic classification when constrained by too little labeled data or insufficient time to train models from scratch. Network traffic classification is essential in network security, network management, and application identification. Labeling network traffic data, however, is often time-consuming and expensive, which limits the amount of labeled data available for training machine learning models.

This thesis investigates using a semi-supervised learning approach that leverages positively labeled and unlabeled data to improve classification performance when faced with a lack of labeled data. The method uses a combination of bootstrap aggregation and tree-based classifiers to classify unlabeled network traffic flows from the same class successfully. This same semi-supervised learning approach also successfully detects zero-day (i.e., never before seen) encrypted messaging applications for which no training data is available.

Additionally, this thesis investigates using deep transfer learning from a state-of-the-art computer vision model for network traffic image classification. By representing network traffic flows as grayscale network traffic images, highly sophisticated image classification models can transfer to the task of network traffic classification. Using these advanced models as a source for training dramatically enhances the speed at which new models can train, addressing the constraint of having too little time for training. To investigate whether deep transfer learning is successful in network traffic image classification, this work used our network flow capture system (which creates a volume of unlabeled data) and commercial appliances (to turn the unlabeled dataset into a real-world labeled dataset).

Experimental results in this thesis demonstrate that the semi-supervised learning technique of positive and unlabeled learning is highly effective at detecting hidden positives amongst unlabeled data. Furthermore, this thesis shows that representing network traffic flows as grayscale images allows state-of-the-art image classification models (e.g., ResNet) to transfer to the domain of network traffic classification effectively.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

At the beginning of this Ph.D. journey, I was working towards the culmination of my education, seeking the highest degree conferred by a university. In the end, it turns out not to be an end at all. Rather, this is the beginning of something new. Before, I had only a vague idea of what I wanted to research. Now, my mind is alight with notions and directions to explore. The lighting of this fire is thanks to those addressed in this acknowledgment.

I want to thank Dr. Tracy Camp for taking a chance on me when she already had far too much on her plate to be my advisor. Her unwavering support and guidance have been invaluable throughout my research journey. Despite her busy schedule and numerous commitments, she always made time for me, patiently answering my questions and providing (extensive) feedback on my work. I feel fortunate to have had the opportunity to work under her mentorship, and I am indebted to her for everything she has done for me.

I also want to express my deep appreciation to Dr. Kerri Stone for her enthusiastic support and guidance throughout my research; I can't imagine this journey without her as an advisor. Dr. Stone consistently demonstrated a deep understanding of the technical aspects of my work, providing insightful feedback that helped me refine my implementation approaches and evaluate their effectiveness. Furthermore, Dr. Stone showed a keen interest in helping me identify meaningful, real-world problems that could benefit from the application of machine learning techniques. Her support was instrumental in shaping the direction of my research and enabling me to produce meaningful contributions to the field.

Finally, I am incredibly grateful to my committee members for their invaluable guidance and support throughout this research journey. Their thoughtful feedback and constructive criticism have challenged me to think more deeply and creatively about the problems I am solving. Their expertise in diverse fields has enriched my research, and I am thankful for their willingness to share their time and knowledge. With their help, I am confident in my ability to contribute to the field of machine learning for network traffic classification. I cannot express enough how much I appreciate their unwavering support and commitment to my success.

To my beloved wife, Whitney, thank you for standing by my side through this long and challenging journey. Any success that is mine is ours, as it was surely built upon your unwavering love, support, and encouragement. Your patience and companionship in moments of frustration and exhaustion have sustained me. I dedicate this Ph.D. to you as a token of my gratitude and appreciation for everything you have done for me. I could not have done this without you.

To my children, Emma, Henry, and Benjamin, I hope this accomplishment inspires you to pursue your dreams and tackle challenging tasks with determination and perseverance. You have been my constant motivation. As you continue to grow and explore the world, remember that anything is possible with hard work, focus, and a passion for learning. May this serve as a reminder always to strive for excellence and never give up on pursuing knowledge and personal growth.

CHAPTER 1

INTRODUCTION

Network traffic classification is the process of identifying and categorizing network data flows based on their characteristics. This field has evolved with changing network environments and technologies. Before the age of ubiquitous encryption in networks and the rise of technologies like network address translation (NAT), traffic could be identified using well-known IP addresses and TCP/UDP port numbers or other techniques, such as deep-packet inspection (DPI). Today, much of the state-of-the-art in traffic classification research utilizes machine learning (ML), which includes deep learning, to learn from more nuanced features of the network flows to accomplish the classification task.

There are several motivations for computer network traffic classification. One of the primary motivations for network traffic classification is to detect and prevent security threats, such as malware, phishing, or denial-of-service attacks. By analyzing network traffic data, security systems can identify suspicious or malicious activity and take appropriate measures to mitigate the threat.

Another motivation for network traffic classification is to optimize the performance and reliability of network services. By prioritizing traffic based on type, quality-of-service (QoS) systems can ensure that critical services, such as voice or video, receive sufficient bandwidth and low latency, while non-critical services, such as email or web browsing, receive lower priority. Network traffic classification can also help network administrators to monitor and troubleshoot network issues. In some cases, network traffic classification is used to measure and bill customers for their network usage. Service providers can calculate usage fees and provide detailed billing statements by tracking the user's volume and type of network traffic.

Identifying applications being used on a network can, however, be concerning for privacy. For example, consider how web browser-based cookies (which track individuals across numerous visited websites and identify the applications individuals regularly use) can help law enforcement, advertising, data broker, or insurance agencies to create profiles about specific individuals. Similarly, network traffic classification can reveal information about what types of applications

individuals are using.

Overall, computer network traffic classification plays a critical role in ensuring the security, performance, and reliability of computer networks. It has a wide range of applications in various industries, such as telecommunications, finance, healthcare, and government. It also has significant privacy concerns attached to it. For these reasons, it is important for researchers to understand to what extent modern classification techniques can effectively classify network traffic in our ever-changing network environments.

In the works presented in this dissertation, we focus on two main challenges to network traffic classification in the modern era. The first is likely the biggest challenge for network traffic classification, and that is obtaining labeled data at a pace that keeps up with the continual release of new applications and updates to existing ones. Applications are constantly evolving due to regular updates, and new applications are emerging all the time. This makes it extremely challenging to maintain up-to-date labeled data for applications of interest, resulting in stale data and, ultimately, degraded network traffic classification performance by classification models.

Semi-supervised learning is a promising approach to reduce the need for labeled data in network traffic classification [1, 2], as it allows the use of a large amount of unlabeled data during model training. Although fully-labeled data is ideal for classification tasks, obtaining it can be costly or impossible. By partially labeling network data, this approach saves time and reduces labeling costs, which can be significant in network traffic classification. This method also helps overcome the issue of class imbalance, where some classes have significantly more labeled data than others, especially for rare or difficult-to-label classes. Overall, training with partially labeled data is a powerful tool in developing efficient and effective machine learning models for network traffic classification that can generalize well to new, unseen data.

We implement a form of semi-supervised learning in Chapter 3 known as postive and unlabeled (PU) learning. We show that by combining unlabeled data with a small amount of positively labeled data for a given class, we can effectively classify mobile network traffic hidden among the unlabeled data. In Chapter 4, we take this technique one step further and train models with a combined class representing a *genre* of applications in order to identify never-before-seen, or *zero-day*, applications.

The constant updating of applications (which creates stale data) leads to the second challenge we focus on in this work: the time it takes to train and deploy the most state-of-the-art deep learning models for the task of network traffic classification. Constant updates to applications and, thus, the labeled data associated with those applications, require retraining of existing or deployment of new models to classify these new applications. Training these new models is a time-consuming process subject to a steady stream of iterations where new models must be routinely trained.

To accelerate the training and deployment of models for network traffic classification, we investigate transfer learning [3, 4] in Chapter 5. Transfer learning is closely related to semi-supervised learning in that both aim to improve a machine learning model's performance in scenarios where labeled data is scarce or expensive to obtain. The goal of transfer learning is to adapt a solution from a source domain to a new target domain. Utilizing information from a source domain can have multiple advantages for training a model in the target domain, such as reduced training time or, like in semi-supervised learning, reduced requirements to have labeled data in the target domain. For network traffic classification, we use transfer learning to pre-train a deep neural network on a large image classification task and then use the pre-trained model (from a different domain, i.e., image classification) as a starting point for the network traffic classification task.

In this dissertation, we begin by describing our data collection systems and methods in Chapter 2. We then discuss work introducing a semi-supervised machine learning technique known as positive and unlabeled learning for classifying mobile applications with only a small amount of labeled data. Building on our PU learning efforts in Chapter 3, we classify zero-day mobile encrypted messaging applications (i.e., those applications that were not a part of the training data and have never been seen by our models) using generalized models built with data from the *genre* of encrypted messaging applications in Chapter 4. Finally, in Chapter 5, we investigate classifying network traffic from a very large real-world dataset using deep transfer learning, which addresses the situation when limited time exists to train new models from scratch. Specifically, we convert network traffic flows into images to be used with state-of-the-art computer vision models with significant success.

Overall, our work shows that semi-supervised learning (e.g. PU learning) and transfer learning can effectively mitigate the two main challenges to network traffic classification: obtaining labeled data at a pace that keeps up with the continual release of new applications and updates to existing ones, and the time it takes to train and deploy the most state-of-the-art deep learning models for the task. Semi-supervised learning and transfer learning are promising lines of research to overcome these challenges and improve the efficiency and effectiveness of network traffic classification in a contemporary network environment.

CHAPTER 2

NETWORK TRAFFIC FLOW COLLECTION FOR MOBILE APPLICATIONS AND CAMPUS AREA NETWORK WI-FI

Large portions of this chapter were published at the 2021 IEEE 29th International Conference on Network Protocols (ICNP) [5] and are reused here in accordance with the IEEE copyright permissions specified in Appendix A.

Jason Hussey[1,2], Ethan Taylor[1,], Kerri Stone[3], Tracy Camp[1,4,5]

2.1 Introduction

Determining the datasets available for exploring a research question is the first step when conducting any machine learning (ML) task. Many publicly available datasets exist for network traffic classification though the quality and represented applications vary greatly. Some of the available datasets are not relevant to mobile network traffic classification, a requirement for our work presented in Chapters 3 and 4, as they either don't contain data generated from mobile devices or it's not clear that they do. Furthermore, these publicly available datasets are often stale, having been created several years ago, which always leaves lingering questions about whether or not research efforts on these public (stale) datasets will translate to real-world (current) data (a topic we explore in Chapter 5). Finally, there's no guarantee that a public dataset will contain sufficient examples of a specific application or genre of applications, limiting the research questions that can be investigated with that dataset.

Another option for researchers is to leverage commercially available datasets. Still, this approach comes with two distinct disadvantages: a cost is typically associated with acquiring a dataset and a non-disclosure agreement (NDA) usually comes with the acquisition. These constraints can often be prohibitive to research teams with insufficient funding or the need to publish the dataset used in their work.

[1]Colorado School of Mines
[2]Primary researcher and author
[3]ICR, inc.
[4]Computing Research Association
[5]Author for correspondence

The third option is to generate a custom dataset, which ensures that the applications represented in a dataset are precisely those that are desired. However, capturing a custom dataset internally also comes with costs, mainly associated with time and capture equipment.

In this work, we have opted for a mix of these strategies. In Chapters 3 and 4, we seek to explore traffic classification for mobile encrypted messaging applications. In [6], a team of researchers provides an open-source dataset containing bi-directional flows (bi-flows) from applications on Android mobile phones. The data is JSON format and comprises traffic from 40 applications on three distinct phones generated by 280 volunteers over a year. While the authors describe the broad techniques used in creating the dataset, the source code is unavailable. In other words, there is insufficient detail about libraries and tools used to capture and process the datasets that would allow other researchers to replicate the process. The inability to create data for new applications is an issue for our work as we seek to explore the application of ML to genres of applications not represented in this dataset. We, therefore, created and implemented a network traffic capture system to approximate the approach in [6].

Finally, we also create a large dataset from real-world traffic captured from a campus network. This capture includes many different applications and is processed into a raw storage format for later use in Chapter 5.

2.2 Mobile Application and Campus Wi-Fi Data Collection

We create two datasets for the work in this dissertation. The first is a fine-grained packet capture of network traffic generated by a *single application* on an Android phone; we capture this dataset using a network packet capture system that we developed similar to the one described in [6]. This custom mobile Android dataset is then used in conjunction with the MIRAGE-19 dataset in [6] in Chapters 3 and 4. For the second dataset, we conduct a week-long packet capture from the Colorado School of Mines' Wi-Fi network, while simultaneously capturing network traffic flow labels from the campus firewall logs. We use this second dataset in Chapter 5.

2.2.1 Android Application Traffic Capture

Capturing network traffic generated by a specific application on a mobile device requires careful system design. The main challenge lies in the complexity of demultiplexing packets

generated by *different applications on the same mobile device*. We reverse engineer the techniques broadly described in [6] to address this challenge. We simultaneously run `tcpdump` on an Ubuntu station serving as a wireless access point and log network system calls from a specified Android application using `strace`[6]. The `strace` log file reveals network socket connections used by the application, allowing us to filter the packet capture down to just those packets. The Android application traffic capture system is illustrated in Figure 2.1. Data generated from this traffic capture system is used in the research described in Chapters 3 and 4.

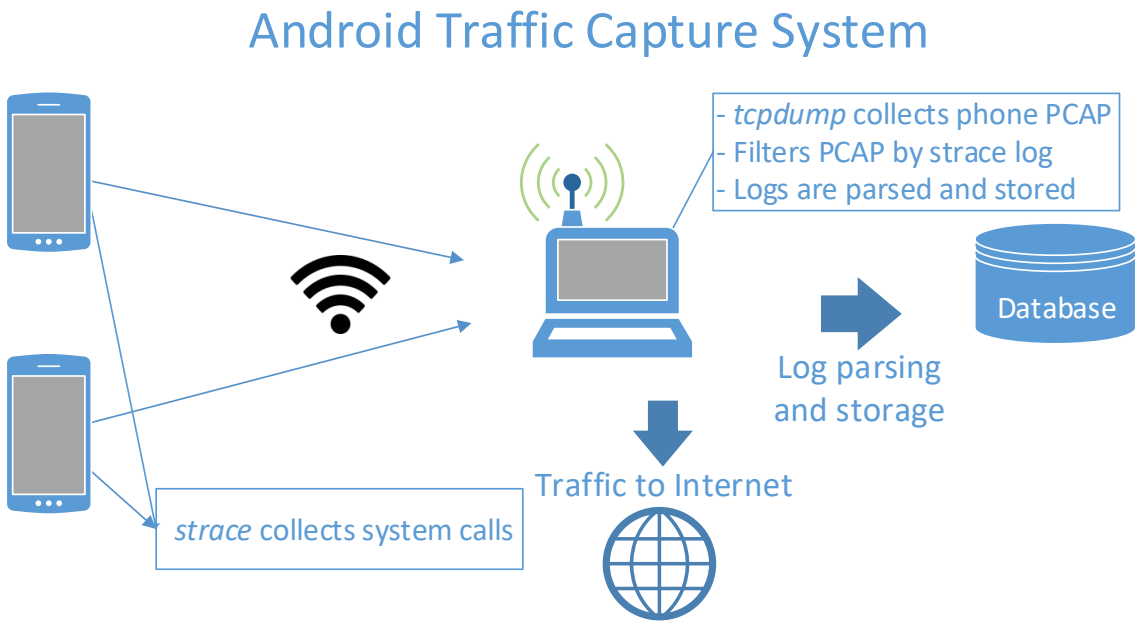


Figure 2.1 Our reverse-engineered capture system for Android network traffic flows. This system was inspired by the work in [6]. Rooted Android phones collect strace logs of network socket calls that are then used to filter the packet captures (PCAP) collected via tcpdump on the capture station. The resultant PCAP is processed and stored in a JSON format.

In our opinion, other techniques for capturing network traffic flows are not as precise. A common technique in the literature is starting a packet capture simultaneously with the beginning of an application's usage. In this scenario, however, there is no way to guarantee that every packet captured belongs to the application of interest. The best outcome would be that the application's traffic is the *preponderance of captured traffic*, and anything else caught in the packet capture amounts to background noise (which, admittedly, might have some benefits for generalization

[6]https://strace.io/

when used in ML training). We take the more precise, but laborious, step toward labeling the application traffic sent from an Android device summarized previously and detailed next.

2.2.2 Campus Wi-Fi Capture

With our institution's (Colorado School of Mines) Information Technology Service, we establish a port mirror that captures the bi-directional Wi-Fi traffic on campus[7]. This mirrored traffic is received on a server within our research team's control, allowing us to process the data further.

The sustained data rate of this traffic is approximately 1.5 Gbps during core hours. To capture this large volume of data without any packet loss, we utilize the ntop[8] organization's `n2disk`[9] utility which allows for multi-gigabit captures. Once a specified amount of data is captured, a file rotation occurs. The current capture file is processed packet-by-packet to save only the relevant information using the `tshark`[10] utility.

The primary challenge with this technique is obtaining labels for the captured packet data. While unlabeled data is the norm in the real-world, not having labels for the flows captured from the campus network makes it difficult to evaluate our ML techniques properly. In many published papers, for example, [7] and [8], researchers use "commercial" datasets provided by ISPs. In these datasets, the labels are typically derived from specialty devices connected to the network that utilize proprietary methods combined with open-source techniques, such as IP addresses, ports, and protocols being used for those applications, that are not tunneled or fully encrypted with SSL yet.

Similarly, we leverage Palo Alto firewalls in the Colorado School of Mines' network to provide labels during our capture session. These firewall devices first attempt to classify traffic by well-known ports and IP addresses, the original method used for network traffic classification. Secondary to this, observable information like certificates exchanged during the TLS handshake is used to categorize traffic. If both methods fail to label a network traffic flow conclusively, the devices use heuristics to analyze the application's behavior and attempt a classification. We simultaneously collect this system log data from the firewalls in addition to our packet capture.

[7]This partnership and implementation were validated and approved by our institution's IRB office.
[8]https://www.ntop.org/about/about-us-2/
[9]https://www.ntop.org/products/traffic-recording-replay/n2disk/
[10]https://www.wireshark.org/docs/man-pages/tshark.html

The campus Wi-Fi capture system is illustrated in Figure 2.2. The resultant dataset is used in Chapter 5.

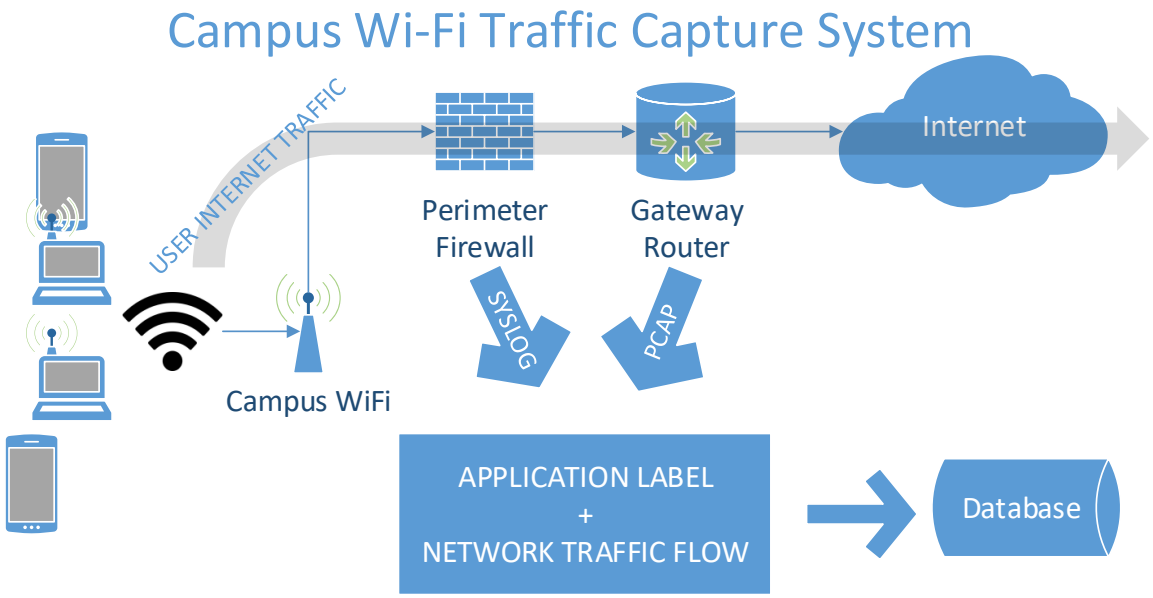


Figure 2.2 This image illustrates our campus Wi-Fi capture system. Network traffic flow labels from the perimeter firewall are captured via SYSLOG and stored in a key-value database. Wi-Fi traffic headers are collected in a packet capture (PCAP) via a port-mirror. Application payload size, direction, and inter-arrival timing values are extracted and assigned a key comprised of the 5-tuple uniquely identifying the flow. This key is then cross-referenced to the captured firewall logs. After combining the application label with the packet data, each flow is stored in a database for later use.

2.3 Experimental Design

For our Android application traffic captures, we generate the data from two models of phones (a Samsung A51 and a Xiaomi Poco X3) running their stock Android OS. While the team in [6] uses a custom OS across all of their phones (e.g., CyanogenMod), we believe the OS variance in our dataset is closer to real-world data. Both phones are rooted to obtain super-user privileges, enabling us to load the `strace` utility onto the phone, generate strace logs for an application during its use, and retrieve those log files from the phone after a capture session is finished.

When capturing packets from the campus Wi-Fi network, we only keep TCP/IP layer 3 and layer 4 headers in each packet. Layer 2 headers are relevant only to the local network and

payloads are largely encrypted today; thus they are both discarded. Excising Layer 2 headers and packet payloads has the bonus of reducing the total disk space required to store the dataset during processing. The captured data is stored locally in PCAP format for further processing described in the next section. For this dataset, we continuously captured data for seven consecutive days.

While packets are being captured, we collect network traffic logs from the Palo Alto firewalls[11] via the SYSLOG utility. This data describes network traffic conversations, so-called "flows" (described in the next section in more detail). Log data collected this way is stored in a key-value database for future lookups. The key, in this case, is the unique identifier for each flow (again, detailed more in the next section) while the value is the label and category that the Palo Alto firewalls assigned the network conversation. There is much more extensive information provided in the firewall log data but, for our purposes, we discard all of it except the application name and category.

2.4 Data Representation

After the collection of our mobile Android data is complete, we begin a global preprocessing of the data that sets the groundwork for all experiments in Chapters 3 and 4. We process (using multiprocessing as the amount of data is extremely large) the packet captures into bi-flows, consistent with much of the traffic classification literature [8]. A bi-flow is a conversation between two hosts where each packet shares the same 5-tuple key containing destination and source IP addresses and port numbers plus the layer 4 protocol. In the case of a bi-flow, the ordering of the source and destination IP addresses does not matter; however, we record the direction of each packet so the re-creation of uni-directional flows is possible with our dataset. We then calculate statistically descriptive features for each network flow and store them in a JSON-formatted file. These statistical features are used for training models primarily in Chapters 3 and 4 and as a point of comparison for our deep transfer learning model in Chapter 5.

For our campus Wi-Fi capture, to create a dataset for the experiments in Chapter 5, the associated packets' timing and size values are stored in arrays for each flow. Aside from being a component of the key field, no IP addressing is carried forward into the analysis, as this would heavily bias results by tying applications to IP addresses. After each packet is sorted and ordered

[11] https://docs.paloaltonetworks.com/pan-os/10-1/pan-os-admin/monitoring/use-syslog-for-monitoring/syslog-field-descriptions/traffic-log-fields

correctly into its flow, we use the 5-tuple key to conduct a database look-up into our captured firewall data. This 5-tuple key is paired with a value containing the label the Palo Alto firewall assigned. These labeled arrays become the basis for all future ML applications with this data. We save the application payload lengths and inter-arrival timings from the first 32 packets from each network traffic flow into a comma-separated value file. That is, each network traffic flow is represented as a single entry in the file by 64 values: 32 for the payloads and 32 for the inter-arrival timings. We note that any TCP control messages (i.e., zero-length application payloads) are discarded as these packets do not describe the behavior of a given application. Each application payload length is an integer signed to represent the direction of the packet, e.g., a negative value indicates the packet's destination was the campus Wi-Fi network and a positive value indicates that the campus Wi-Fi network was the source of the packet. Inter-arrival values are stored in microseconds. From this per-packet data associated with each 5-tuple key, we can generate various statistics about the flow, including features such as total bytes sent, the total length of the flow, and data rates. Further details about preprocessing of this captured data are provided in Chapter 5. Additionally, in Chapter 5 we discuss how we turn this raw packet data into *network traffic images* to be used in classification with computer vision models.

2.5 Acknowledgment

We thank Colin Randall, George Hendricks, and Dr. Phil Romig of Information Technology Services at Colorado School of Mines. They were each instrumental in the Wi-Fi data collection and the navigation of and compliance with institutional policies. Additional thanks to Dr. Romig for review of our Android capture system.

CHAPTER 3

POSITIVE AND UNLABELED LEARNING FOR MOBILE APPLICATION TRAFFIC CLASSIFICATION

A shortened version of this chapter was published at the 2022 IEEE Military Communications Conference (MILCOM) [9] and is reused here in accordance with the IEEE copyright permissions specified in Appendix A.

Jason Hussey[12,13], Kerri Stone[14], Tracy Camp[12,15,16]

3.1 Introduction

Network traffic classification is an established field in computer science, though the techniques used to perform the task have evolved over the past decades [10, 11]. Well-known IP addresses and TCP/UDP port numbers were the prominent and initial features used to classify network traffic. As these features became less informative and network security requirements grew more complex, a more intrusive interrogation of TCP/IP traffic, by way of deep packet inspection (DPI), was proposed. As the share of encrypted application payloads rose, deep packet inspection's effectiveness similarly declined [12, 13]. Without conducting even more intrusive forms of examination, such as SSL intercept, where the payload's contents are de-encrypted and inspected, internet service providers (ISP), corporate IT departments, and other interested entities have fewer viable tools for traffic classification tasks. Recently, network traffic classification has been tackled using machine learning (ML) (including deep learning), which can classify traffic based on various features. Though statistical methods have always existed for network traffic classification, their popularity rose dramatically as more approachable (and less computationally expensive) options fell out of favor.

The applications of network traffic classification are just as varied as the techniques to conduct the classification. The quintessential and oft-cited example is that of an ISP seeking to implement

[12]Colorado School of Mines
[13]Primary researcher and author
[14]ICR, inc.
[15]Computing Research Association
[16]Author for correspondence

what is known as Quality-of-Service (QoS) in their networks. That is, they seek to prioritize certain types of traffic over others, typically aimed at ensuring real-time communications are not disrupted. Other research in traffic classification is applied towards intrusion detection, defeating malware, and other cybersecurity threats on a network where the first step in defeating the threat is detecting it. Less discussed is the application of traffic classification to establish patterns in the behavior of users on the network, which can have a significant impact on user privacy. Such information about users would be relevant to any number of entities, including advertisers, criminal entities, or law enforcement organizations.

Mobile network traffic classification, which is the subject of this chapter, focuses on the network traffic generated by applications on smartphones, i.e., Android or Apple iOS phones. One of the main challenges to leveraging supervised machine learning techniques for mobile application classification tasks is obtaining fully labeled datasets to train models. It can be costly, if not impossible, to obtain such data in the real world. Most often, researchers rely on public datasets created several years prior that contain a fixed number of applications. If a research team only cares about demonstrating in the abstract that a technique is useful for a classification task, these datasets are sufficient. However, if the researchers wish to investigate whether *specific applications* are able to be classified effectively, they would be unable to do so without creating a new dataset or expanding an existing one with traffic from the applications of interest. Moreover, network traffic patterns for mobile applications are known to change over time as new features and updates are released.

To emphasize privacy concerns, imagine an organization captures network traffic on a given network that has users communicating via an encrypted messaging application, such as Signal.[17] Encrypted messaging applications afford users privacy for the contents of their messages, but are susceptible to side-channel attacks that take advantage of information leaked through network traces. Instead of identifying every traffic flow on this network, suppose the organization only cares about when Signal traffic appears. Not identifying every flow enables the organization to build a relatively small, labeled dataset for the Signal application. Then, a model is trained with those positive examples and rapidly deployed against the larger unlabeled dataset, identifying only the target application's presence. In this chapter, we explore this topic further.

[17]Signal Messenger: Speak Freely @ https://signal.org/en/index.html

The main contributions of this chapter are:

- Demonstrating that positive and unlabeled learning, a technique requiring only a small set of labeled samples for a target mobile application, is effective at classifying Android applications, including encrypted messaging applications, such as Signal. Specifically, we attain an average f1-score of 0.8708 when classifying Signal in this way and an average f1-score of 0.6497 across every application in our dataset.
- Providing proof that it is possible (and easy) to classify a target application amongst a large unlabeled dataset using only a small set of data labeled for that application, which is a significant security risk.
- Contributing to the feature selection discussion for bi-directional network traffic flows of mobile applications.

3.2 Overview

We use semi-supervised learning techniques to address the problem of an organization wanting to detect just a single class of mobile application traffic in a large, unlabeled dataset. In particular, we leverage one-class classification (OCC) and the closely related technique of positive and unlabeled (PU) learning. In OCC and PU Learning, we have access to a small set of data for the positive class, P, and a more extensive set of unlabeled data, U. In U, we know there are many hidden instances of the positive class, likely exceeding the number of samples we have labeled in P. In other words, we have some examples of the desired application to train with, but know that there are many more instances amongst the unlabeled data that we want to classify.

We explore OCC and PU Learning on a fully-labeled dataset extended with our Signal application traffic from Android phones. Iterating through each of the 18 applications in our dataset, we give every application a turn as our one positive class, ultimately creating 18 different classifiers. After selecting the positive class in a given round, we remove the rest of the labels in the dataset for that iteration; in other words, we create a large unlabeled dataset similar to the real-world problem where obtaining fully labeled data is infeasible. Next, we hide a portion of the current positive class in the unlabeled data. Removing the labels from a fully-labeled dataset during training and prediction has an advantage when it comes to evaluation, i.e., we know the

original labels with which to score our results. By evaluating our approach on a fully-labeled dataset, we build confidence in the technique for the eventual application to data generated in a real-world environment for which we will likely not have labels.

OCC, discussed in more detail in Section 3.5, does not leverage any unlabeled information. PU Learning, on the other hand, takes advantage of the large unlabeled dataset and the information contained within. We are aware of three broad approaches to PU learning: binary classification (where we naively treat all unlabeled data as negative during training) [14], bootstrap aggregating (also known as 'bagging') [15, 16], and a 'two-step' method [17, 18]. We present a brief overview of these techniques in the next few paragraphs.

The first approach is similar to a binary classification, where all of the unlabeled data is fed to the model as negative training examples. Using this approach is problematic because we know there are positive samples in our unlabeled data (the entire motivation for trying to train models to detect them). We confuse the model and lower its prediction accuracy by telling the model that all unlabeled data are negative when some are, in fact, positive.

The second technique, bootstrap aggregation, is a type of model averaging for PU learning [15, 16]. Bagging involves many iterations where a uniformly random subset of the unlabeled data is sampled with replacement and given to the model as negative examples. Additionally, the model trains on all positive samples in every iteration. In other words, a model is provided the same positive labels with some new, random samples from the unlabeled data serving as temporary negatives for each round of training. After training in an iteration, the model is used to provide a prediction probability on whether the *out-of-bag* samples, that is, the unchosen samples, belong to the positive class. These out-of-bag predictions are averaged throughout many iterations to obtain an average predictive probability for every sample in the unlabeled data. We emphasize that the model only provides a probability of belonging to the positive class, not a label for each sample.

Looking at a toy example for clarity, suppose we have an unlabeled sample randomly chosen to be negative in the first two training iterations of the model and then not chosen in the next five training iterations. This sample does not receive an out-of-bag prediction in the first two iterations. In the last five iterations, suppose the classifier assigned a probability of belonging to the positive class of $p = 0.86, 0.75, 0.34, 0.48, 0.52$, respectively. These individual out-of-bag scores

are averaged, giving us a prediction probability of $p = (0.86 + 0.75 + 0.34 + 0.48 + 0.52)/5 = 0.59$. Assuming a standard prediction threshold of 0.5, we would assign this unlabeled sample as $class = 1$, given that $p > 0.5$ in this example. We note that any average out-of-bag score so close to the prediction threshold will be more likely to result in an error, something we explore in the PU learning results section of this chapter (see Section 3.6.3).

The third approach, a so-called 'two-step' method [17, 18], requires an initial attempt to identify 'reliable negatives' from the unlabeled dataset. Several techniques exist to choose these reliably negative samples, including clustering and classification, and they all first identify a few samples that are highly unlikely to be positive, the *reliable negatives.* The second step of this 'two-step' method is training a traditional classifier with the known positives and newly identified reliable negatives. This classifier then makes predictions for the unlabeled data points, identifying a new set of negatives to move from the unlabeled set to the negative set based on a researcher-defined prediction threshold. This process continues until no samples fall below the chosen threshold for classification as a *reliable negative*. With no more negative samples to extract, one proceeds with the final training and classification to obtain the model's prediction as to whether or not the remaining unlabeled samples are positive.

In our study, we implement *bagging*, discussed further in Section 3.6, as we found this approach to provide more consistent results across our dataset. Before we implement either OCC or PU learning, we first explore the ability of our dataset to be classified using multi-class classification. As the PU learning approach utilizes more available information than the OCC technique, but not as much as the fully-labeled multi-class classification, we anticipate that the order of performance will be multi-class classification > positive and unlabeled learning > one-class classification.

3.3 Data Acquisition

We use a dataset comprised of two key components in this study. The first component is a public dataset, named MIRAGE-19, provided by the researchers in [6]. This dataset contains bi-directional network traffic flows (hereafter, bi-flows) and is discussed further in the following sub-section. The second component is essentially an extension of the first, where we created infrastructure (see Chapter 2) to implement the same bi-flow capture capability. Having a reverse-engineered capture system allowed us to extend the public dataset from [6] with traffic

from another app, i.e., the Signal application.

3.3.1 MIRAGE-19 Dataset

As discussed in Chapter 2, the research team used 280 human volunteers to generate traffic from 40 applications on three discrete Android phones [6]. The `strace` utility was used to identify the packets associated with a given application, which provides logged output detailing every network socket call made by the application. These identified packets were then captured and parsed into multiple JSON-format files containing various features about the given bi-flow. Due to the nature of some mobile applications generating significantly more data than others, this dataset is inherently imbalanced. In other words, the applications represented have a widely varying number of traffic flows.

Table 3.1 This table provides the per flow features (102 total), each calculated for (1) two attributes (payload size and inter-arrival timing), and (2) three directions (upstream, downstream, and bi-directional).

Feature Name	Directions × Attributes	Number of Features	Feature Description
min	3×2	6	Minimum
max	3×2	6	Maximum
mean	3×2	6	Average
std	3×2	6	Standard Deviation
var	3×2	6	Variance
mad	3×2	6	Mean Absolute Deviation
skew	3×2	6	Skewness
kurtosis	3×2	6	Fisher Kurtosis
x_percentile	3×18	54	Value at each of 9 deciles (10 through 90)
Total		102	Sum of all features

We select a subset of the available features in the dataset from [6], removing all of the *per packet* features, as we do not use any vector features for our models. However, these *per packet* features are necessary if one desires to engineer new features for this dataset. Table 3.1 and Table 3.2 summarize the statistical features of the flows we chose. As part of our pre-processing, we rename and combine these features into a single JSON object per-flow to more easily load them into a data frame for our ML model training. When complete, we have a total of 114

features per-flow available for consideration, not including the label: (4 metadata features $*$ 3 directions) + (17 statistical features $*$ 3 directions $*$ 2 packet features) = 114 features.

Table 3.2 This table provides the per flow metadata attributes calculated for three directions (upstream, downstream, and bi-directional). There are 13 total features (12 statistical features + 1 label).

Feature Name	Directions × Attributes	Number of Features	Feature Description
num_packets	3×1	3	Minimum
IP_packet_bytes	3×1	3	Maximum
L4_payload_bytes	3×1	3	Average
duration	3×1	3	Standard Deviation
BF_label	N/A	1	Application label for bi-flow
Total		13	12 features + 1 label

3.3.2 Encrypted Messaging Applications

In addition to the applications captured in the MIRAGE-19 dataset, we seek to explore how well network traffic classification techniques work on the genre of encrypted messaging applications, e.g., the Signal messaging application. As discussed in Chapter 2, we closely studied the MIRAGE-19 dataset details and re-implemented the system described in [6] by reverse engineering their capture pipeline. We recreate much of their technique as, while some key insights are provided in [6] to describe their technology and design choices, no source code is available. Figure 2.1 from Chapter 2 provides a visual depiction of our capture system.

We note the network environment with which we captured our data likely has different characteristics than the network with which the original researchers captured data. Despite an identical setup (insofar as we can discern from their paper), numerous variables likely differ, such as the RF environment, upstream network performance, and internal phone hardware. The most natural way for these differences to manifest is in the inter-arrival time features, which measure the time between packets in a particular stream of traffic. We discuss inter-arrival timing features further in Section 3.3.3.

Another difference concerns the duration of captures. While most network traffic flows do not last the entire capture session, there is a possibility that some flows generated on our capture

system run the entire session length. If the number of these flows was high and the duration was constant, they could be easily identifiable by that feature. In order to address this issue, we randomly chose the length of a capture session to be between 5 and 15 minutes, aligning with the duration of capture sessions in [6]. Since a very small percentage of flows run the entire capture duration, randomizing our capture length should be enough to mitigate any potential bias.

Using our reverse-engineered Android application capture system, we captured 1,325 human-generated, bi-directional flows from two rooted Android phones for the Signal messaging application. Specifically, we used a Samsung A51 and a Xiaomi Poco X3. Both phones run version 10 of the Android OS and 5.25.7 of the Signal app. Rather than synthetic interactions, human users generated all of our flows to create the most realistic dataset possible. The types of activities our human users performed when generating our labeled Signal data ran the gamut of 'everyday operations,' i.e., one-to-one text messaging, file sharing, group chats, account setup and management, as well as video and voice calling. These activities occurred between phones where each phone participating was connected to one of three possible network environments: the upstream campus network, our capture system's access point, or a cellular network in the nearby geographic vicinity.

3.3.3 Feature Selection

This section discusses some points of consideration when deriving a feature set to use for classification tasks in subsequent sections. The number of features to describe a network traffic flow is immense and can be very nuanced. Many decisions await researchers looking at this problem, which can impact results and performance based on those choices.

For this study, consistent with much of the literature, we describe a bi-directional network traffic flow as the interleaving of the traffic flows traveling from one host to another host on the internet. That is, there is an *upstream* and a *downstream* series of TCP/IP packets. In this case, *upstream* are the packets traveling away from a capture device and *downstream* are the packets with a destination on a capture device.

As described in Section 3.3.1, we use a dataset provided by another research team as well as an extension to this dataset that we created in our network environment. Samples we created and stored contain features describing the individual, bi-directional network traffic flows for both their

packet lengths and inter-arrival times.

We remove all inter-arrival time features from our dataset for three reasons. First, because our network environment is different from that of the primary dataset we utilize, the inter-arrival timings are likely different in our extension to the dataset, which makes classification results biased and difficult to interpret correctly. Second, there is existing literature to suggest that inter-arrival times are not a discriminating feature [19]. Lastly, the curse of dimensionality comes into play, where more features used in model training result in longer computation times and make the models more prone to over-fitting.

Before deciding to drop all inter-arrival time features, we examined the effect of inter-arrival features on our dataset and learned that there was little to no impact on our evaluation metrics when we removed them from the training set. We noticed a slightly above average increase in our Signal traffic results when including inter-arrival time features, but the increase was not significant. Thus, we elected to exclude all inter-arrival time features from our dataset.

In [20], the authors present an analysis of discriminating features for network traffic classification. Their work noted that "packet size and port information always yield the highest accuracies." That port numbers are so effective is no surprise. Our study purposefully ignores port numbers as they can be obfuscated either intentionally by way of VPN or the like, or unintentionally as modern network applications do not always leverage well-known port numbers. Our study similarly observes that the packet size-related features are the most important. In other words, consistent with the findings in [20], we constrained our features to statistics concerning flow duration and the size, both by packet and cumulatively.

The researchers in both [6] and [21] leverage a series of descriptive statistical features during classification tasks, such as skewness, kurtosis, medians, minimums, and percentiles. We similarly include these features in our dataset. Unlike both of these works, we elected not to include timing-related features. We also avoid some of the more complex features derived from packet lengths in [21], such as variances in total bytes over different quartiles of the entire network traffic flow. In our early experimentation, these features are highly correlated with others and do not significantly improve prediction results.

3.4 Multi-Class Classification

In order to establish a baseline for comparison, we implement a typical multi-class classification against the fully-labeled MIRAGE-19 dataset with our Signal application extension. In other words, before we proceed with learning tasks on our data in an unlabeled state, we use the labeled data to determine the best possible performance given all the information we have.

3.4.1 Overview of Multi-Class Classification

Many efforts have focused on performing multi-class classification, including [20–23]. In a recent study, the authors of [24] show promising results with extreme gradient boosting (XGBoost)[25] for multi-class classification on a dataset gathered from a French ISP. Gradient boosting is an ensemble method that combines many weak learners to form a strong learner by successively training models to improve on the errors of the previous model. Notably, the features used in [24] are much more extensive than the coarse features we use. For example, the authors utilize the port and protocol, a strong indicator of the application in question. We purposefully withhold this data from training, and only use it as part of the unique 5-tuple to identify a flow. Nevertheless, to demonstrate a baseline for classification of the extended MIRAGE-19 dataset in its fully labeled form, we use the XGBoost classification technique, albeit on a simpler set of features.

During preprocessing, we drop any class in the dataset with less than 1,000 samples. After dropping these under-represented classes, 18 classes remain with a total of 97,638 samples. Dropping classes with few samples helps make the results easier to interpret and avoids skewed performance due to too few training samples. Our algorithm splits the data into 70% training and 30% testing before fitting the classifier. Because XGBoost is a tree-based classifier, feature standardization is not required and foregone in this instance.

We conduct two different classifications to gauge how feature selection impacts the ability to classify our 18 applications. First, we only use the metadata features about each flow. As previously mentioned in Section 3.3.1, our metadata features include the total number of packets, the duration in seconds, the total payload bytes, and the total IP packet sizes (payload bytes + layer 3 and 4 header bytes). Each of these metrics is recorded for the upstream, downstream, and bi-directional interleaving of the two, providing 12 metadata features

(3 directions $\times$ 4 $\frac{\text{features}}{\text{direction}}$ = 12 features). Second, we fit the classifier on 51 statistically descriptive packet length features (17 $\frac{\text{features}}{\text{direction}}$ $\times$ 3 directions = 51 features) for a total of 63 features. As noted in Section 3.3.3, inter-arrival timing is removed to avoid any bias introduced by our capture system developed to extend the MIRAGE-19 dataset.

To provide a holistic evaluation of this classification task, we calculate the accuracy, Matthew's correlation coefficient (MCC), weighted f1-score (the weighting is derived from the number of true instances of the label in the test data), log-loss, and the receiver operating characteristic area-under-the-curve (ROC-AUC) score to account for our imbalanced dataset[18]. Reported scores are the averaged result of 1,000 iterations, using a different training/testing split for each.

3.4.2 Multi-Class Results

Using the XGBoost classifier with all of its default parameters in `scikit-learn` [26], we attain an average f1-score of 0.7791 for all of the classes combined, a ROC-AUC of 0.9787, and a log-loss of 0.6960. Since we seek to demonstrate what an off-the-shelf classifier can do, we used the default hyper-parameters of the XGBoost classifier; we would, of course, expect improved performance with proper tuning. We are interested in determining to what degree encrypted messaging applications are classifiable; thus, we note that samples from the Signal application have the highest average f1-score of 0.9685.

Statistical results are shown in Table 3.3 and the individual application scores are shown in Table 3.4. We note that Table 3.4 shows the Android package names; the Signal messaging application is denoted as "org.thoughtcrime.securesms." A box plot depicting the distribution of the individual applications' precision, recall, and f1-scores is in Figure 3.1. In this plot, we note that the multi-class classification f1-scores has the highest average median compared with the subsequent technique we discuss in the next section. That multi-class classification with fully-labeled data appears to perform better than our more constrained approaches is expected, as this technique has more information available for predictions[19]. Lastly, given that the addition of the 51 statistical features to our metadata features improves performance so significantly (see

[18] Accuracy can be sensitive to imbalanced datasets when techniques such as under- and over-sampling are not implemented to overcome the effects of imbalance. Nevertheless, we provide the accuracy score here for completeness. Though this particular dataset is not *heavily imbalanced*, it can still prevent a straightforward interpretation of accuracy. We also calculate MCC, f1-score, and ROC-AUC, which provide better intuition into an imbalanced multi-class classification model's performance.

[19] For example, note the high precision results in Table 3.5 for multi-class classification.

Table 3.3), we use all 63 combined features in the next two experiments.

Table 3.3 Average results from all applications using XGBoost multi-class classification with only 12 metadata features. Those metadata features are combined with 51 features describing packet lengths in the flows, for a total of 63 features.

	XGBoost classifier trained on 12 metadata features		XGBoost classifier trained on 63 statistical features	
Metric	Training Set	Testing Set	Training Set	Testing Set
accuracy	0.8127	0.6672	0.9301	0.8013
f1-score	0.8129	0.6653	0.9301	0.7791
ROC-AUC	0.9847	0.9412	0.9976	0.9787
MCC	0.7983	0.6411	0.9248	0.7860
log loss	0.7580	1.1653	0.3406	0.6960

Table 3.4 This table shows f1-scores on the testing dataset averaged across 1,000 runs for each of the three techniques presented in this study. Results for individual applications are listed by their Android package names.

	Multi-class f1-score	PU Learning f1-score	OCC f1-score
Average	0.7791	0.6497	0.3290
Maximum	0.9685	0.9299	0.7410
Minimum	0.5628	0.3342	0.1009
air.com.hypah.io.slither	0.7556	0.6624	0.2774
com.accuweather.android	0.8273	0.6562	0.4159
com.android.vending	0.7242	0.5866	0.3044
com.contextlogic.wish	0.8023	0.6913	0.2528
com.dropbox.android	0.7006	0.6262	0.3474
com.duolingo	0.8145	0.6665	0.3323
com.facebook.katana	0.6877	0.5453	0.1783
com.facebook.orca	0.6863	0.5088	0.1663
com.google.android.youtube	0.7746	0.6708	0.2321
com.iconology.comics	0.8589	0.7418	0.3990
com.joelapenna.foursquared	0.7777	0.6085	0.2845
com.spotify.music	0.8601	0.7613	0.3393
com.tripadvisor.tripadvisor	0.5628	0.3342	0.1009
com.twitter.android	0.6687	0.4844	0.1747
com.waze	0.9619	0.9299	0.7363
de.montain.iliga	0.7914	0.6588	0.2983
it.subito	0.8002	0.6905	0.3403
org.thoughtcrime.securesms	0.9685	0.8708	0.7410

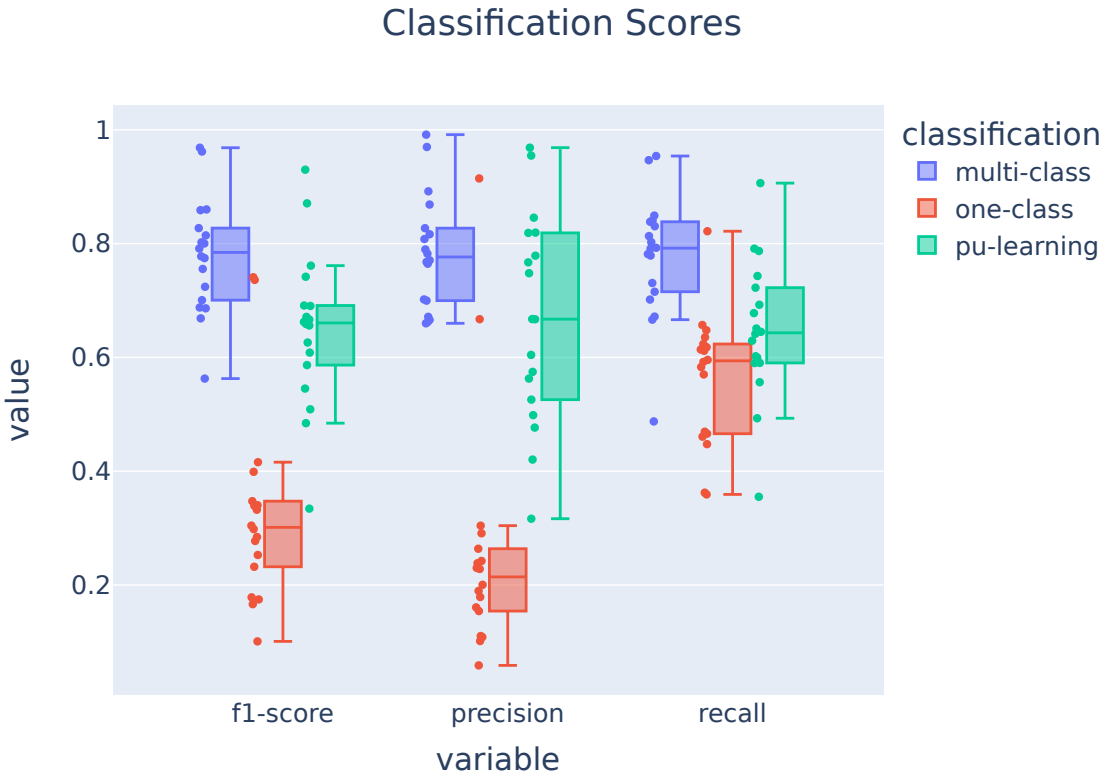


Figure 3.1 Classification results for all techniques and applications discussed in this chapter are shown as a box plot to illustrate the distribution of the unweighted f1-scores, precision, and recall. As hypothesized, using full information for multi-class classification has the highest average median f1-scores, followed by PU learning, where we leverage the unlabeled data, followed by one-class classification, where we do not leverage the unlabeled data.

3.5 One-Class Classification

In *one-class classification* (OCC), we have a relatively small set of samples belonging to the positive class and a large set of unlabeled data, which we expect has a significant amount of positive samples (though we do not know for sure). Anomaly detection is a typical application for OCC. We use OCC to answer the question: given a dataset of all 'positive' samples, does a new unknown sample belong to the dataset or not?

As previously discussed, in OCC we ignore the samples belonging to classes other than our current positive class. Every other sample is treated as an unknown, unlabeled network traffic flow and there are no negative samples provided to the models during training. This constraint

mimics real-world applications where it is too complicated, too time-consuming, or outright impossible to label a dataset fully.

3.5.1 Overview of One-Class Classification

A few techniques are common in one-class learning, e.g., the One-Class Support Vector Machine (SVM) [27] and a tree-based method known as Isolation Forest [28, 29]. Fortunately for researchers, `scikit-learn` provides both of these techniques, along with a few others, in its libraries. As we previously utilized a tree-based method for multi-class classification, we considered using isolation forests for OCC. However, our preliminary experimentation showed one-class SVM outperforms isolation forests in most situations. These results were not unexpected as researchers in [29] note that isolation forests perform best when they "are the minority consisting of few instances" and "have attribute-values that are very different from those of normal instances." While we satisfy the first condition, given the nature of network traffic flows and the very coarse features we use, each application's feature values are not exceedingly different from the others.

The sentiment that network traffic flows share many properties and are hard to differentiate is the precise motivation for network traffic classification. Despite outperforming isolation forests in our tests, even one-class SVMs struggle with several of our classes. One-class SVM performance is also discussed in [30] where researchers compared their *FlowCop* framework, which combines many one-class classifiers, against the `scikit-learn` provided `OneClassSVM` model. The researchers in this paper note that "... the principle of SVM is to construct a boundary in the feature space, which can separate samples of two classes well." Because our "second class" in this instance comprises network traffic flows from many applications (i.e., many classes), it is difficult to separate them accurately.

We proceed with one-class SVM to demonstrate how classification results degrade when models do not utilize negative samples. While impressive results are possible with one-class classification for certain applications (despite having no explicit negative examples available to the models), the data we have not marked "positive" contains information that can aid classification.

Since we expect OCC to perform poorly, compared to the other two techniques, we leverage hyper-parameter tuning for each application to obtain the best model *per application*. As

described in Section 3.2, we take our extended, fully-labeled dataset and iterate through each application. First, we split the dataset into stratified training and testing data, following the same 70%/30% split discussed in Section 3.4. We then set a label of 1 for each sample in our data belonging to the *currently positive* application. The remaining samples are then assigned a label of -1 representing "unknown." This temporary class, -1, is comprised of the traffic flows from the 17 other applications currently not labeled as the positive class ($class = 1$). Finally, a one-class SVM is trained on those samples in the training data marked as $class = 1$ and classifies the samples in the testing data as positive ($class = 1$) or unknown ($class = -1$).

The feature set provided to the one-class SVM during training is the same full feature set as in Section 3.4. We use all of the metadata features in addition to the descriptive statistic features for packet lengths outlined in Table 3.1 and Table 3.2, noting once again that we do not leverage any inter-arrival timing.

3.5.2 One-Class Results

Because OCC utilizes the least amount of information when training models, we expect the results to be the lowest of all three experiments. The average f1-score for all of the one-class SVM models combined is a mere 0.3290 with the maximum f1-score (0.7410) coming from our encrypted messaging application, Signal, and the minimum f1-score (0.1009) belonging to TripAdvisor, a travel-related application. Compared to that of multi-class classification, OCC's average f1-score is 45 percentage points lower. Figure 3.1 depicts the distribution of the individually trained one-class SVM models' results alongside that of the other approaches in this chapter, while Table 3.4 lists each individual application's averaged f1-score for one-class classification.

Similar to the results in multi-class classification, we note that our encrypted messaging application, Signal, shows a strong ability to be classified. Using OCC, the Signal application has the highest average f1-score of 0.7410 across 1,000 training iterations. While various parameters can yield different false positive rates, we note that our optimized one-class SVM predicts $class = 1$ for the Signal messaging application correctly 91.46% of the time (precision = 0.9146, as shown in Table 3.5 on page 31).

3.6 Positive and Unlabeled (PU) Learning

Previously, we investigated traditional multi-class classification and then constrained the dataset to just one class positively labeled. In one-class classification, we removed the label from all samples that did not belong to the one positive class and did not use the unlabeled data in training. While counterintuitive, valuable information in the unlabeled dataset exists. This section discusses our bagging implementation of positive and unlabeled learning for classification on our dataset.

3.6.1 Overview of PU Learning

Positive and unlabeled learning [14, 15, 17] is a form of semi-supervised learning. With PU learning, our goal is to improve upon one-class classification by using information in our unlabeled dataset, U, to detect the set of positive samples, P. While we have ground truth on all of our data, we withhold information to mimic what it would be like in the real world, when there is a large but unlabeled dataset and only a small but labeled dataset for the application we are interested in predicting. The known labels for each sample are then revealed after classification to evaluate our model's ability to detect the application of interest.

3.6.2 Implementation

When using a bagging approach to PU learning, we must first decide which underlying classifier to use for predictions. We experimented with the `sklearn`-provided support vector machines for classification (after scaling the features), decision tree classifiers, random forest classifiers, as well as the extreme gradient boost (XGBoost) classifier discussed in Section 3.4. The XGBoost classifier outperformed every other classifier across all applications in our analysis. We, therefore, choose this method for all our PU learning experiments.

Similar to the methodology in Section 3.5, we select the 18 classes that have more than 1,000 samples in our dataset. To assess the performance of PU learning on our dataset, we repeat the entire PU learning classification process for each application, giving each one a turn at being the single positive class in our dataset. More formally, we have a set of positive classes $P = \{P_1, P_2, ..., P_{18}\}$ where P_i is all samples belonging to the i-th class. For each unique set of positive samples, P_i, we remove the labels from the samples belonging to every other class,

forming the unlabeled set, U_i.

After choosing a positive class, we hide most of its samples in the unlabeled dataset (by changing their labels from positive to unlabeled). The percentage of positive samples hidden in the unlabeled dataset is a parameter that is tunable. We note it is helpful to have an approximation of the percentage of unknown samples that we expect to belong to the positive class. While this percent could vary wildly, it is reasonable to expect that no single application would be more than 10% of traffic on a real network, and most likely it would be much lower.

Most of the results presented are with 90% of a given class hidden in the unlabeled data, leaving us with just 10% of the positive samples to use during training. As an example, hiding 90% of the Signal application results in 1,192 samples being hidden ($floor(1,325 \text{ Signal flows} \times 0.9 = 1,192)$), where 1,192 samples is 1.2% $(1,192/(97,638 - 133))$[20] of all the unlabeled data. We note Signal is actually our least populated class. The percentage of hidden positives ranges from 1.2% (for Signal) to 10.6% (for our most prevalent application, Waze). We increased the percentage of hidden samples to 95% and then 99%, to assess how the results changed when fewer positive instances were available for training.

After selecting our positive training samples, we begin training with our bagging technique. We iterate 1,000 times to create a robust out-of-bag score, randomly selecting N (a tunable parameter) unlabeled samples as negative training instances in each iteration. These N points combine with the 10% of positive samples that we did not unlabel to form our training data for a given round. In other words, the same 10% of positive samples plus a new set of negative samples serve as the training data in every training round. In each round, the temporarily negative training examples do not receive a prediction; however, the samples left out-of-bag (not chosen to be negative) get a prediction probability, and then the iteration ends.

The number of negative training examples, N, selected for each training round is also a tunable parameter. In a balanced dataset, N would be set to the number of positive examples we use in training; in an imbalanced dataset like ours, one might select many more negative examples (since we suspect the number of samples belonging to the class is a small percentage of the total network traffic data). Our experimentation found that selecting N to be between 1% and 4% of

[20] 1,192 is the number of Signal samples we are leaving hidden in the unlabeled data. 133 Signal samples are our *positively labeled samples* in this example. Thus, we subtract those 133 from our 97,638.

the unlabeled data for each application yielded the best performance.

3.6.3 Positive and Unlabeled Results

When evaluating the classification performance of a model, the default prediction threshold is typically 0.5. This prediction threshold forces the model to choose a label for every sample, which leads to increased false positives and a general decline in performance. If, instead, we have an estimation of how many hidden positives we expect to find in the unlabeled data, we can achieve better results. In other words, setting the prediction threshold to obtain roughly the number of positive predictions that we expect to find will generally improve the classification performance. For instance, assume that 1,200 positive samples are hiding in an unlabeled dataset, but our model classifies 3,500 samples as positive. In this scenario, we would see improved performance (primarily fewer false positives) if we increase the prediction threshold and classify fewer samples as positive.

Furthermore, if we sort our model's prediction probabilities for the samples and take only the 100 most likely to be positive, the percentage of those 100 that are, in fact, positive is quite high. Figure 3.2 visually depicts these results, which vary for each application. This discussion underscores the importance of having an estimation on the number of positives hidden in the unlabeled set with PU learning.

To evaluate the performance of our PU learning approach, we plot each application's results on a line graph. Figure 3.2 shows these results. We discuss our Signal application, labeled by its Android package name *org.thoughtcrime.securesms*, to illustrate the meaning of the results in the figure. The Signal application had 1,192 samples 'hidden' in the unlabeled dataset. When we order the out-of-bag scores for Signal from highest to lowest, i.e., *most likely* to *least likely*, and take the top 119 (10% of our expected hidden size, ($floor(1192 \times 0.1) = 119$), the accuracy is 100% (i.e., all of the top 119 samples are, in fact, hidden positive samples). As we increase the number of samples that we label as positive in the unlabeled dataset, we see the model's accuracy start to decline. In Signal's case, when we label the top-X predictions as positive, where X is equal to 70% of the hidden population ($X = floor(1,192 * 0.7) = 834$), the accuracy declines to 99.2%. Finally, when we label the exact number of predictions that we expect to find in the unlabeled set, i.e., 1,192, Signal's accuracy drops to 89.0%.

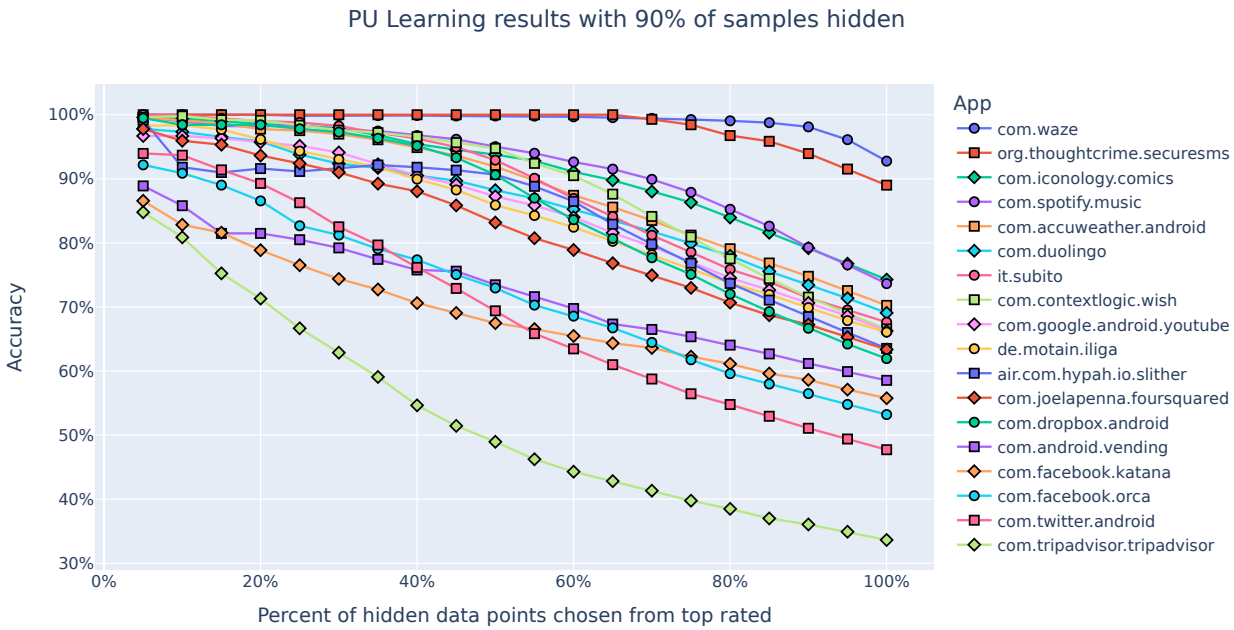


Figure 3.2 Positive and unlabeled learning results after giving each application a turn at being the positive class. The legend is ordered from top to bottom by performance results for each class. The x-axis represents the number of predictions the classifier was asked to make based on the total known to be hidden in the unlabeled set. The x-axis is scaled as each class has a different number of samples hidden. The y-axis depicts standard accuracy. Because the variance of our reported accuracy is very low for each of these data points, we forego displaying error bars to keep the visualization cleaner.

Because our technique involves training many classifiers and combining their out-of-bag scores into an overall average prediction probability, we do not generate clean, crisp labels. We can, however, manually specify a prediction threshold and generate the predictions ourselves. That is, we can assign $class = 1$ to any sample where the $score > 0.5$. Our observations are that for any underrepresented class, such as Signal, which comprises just over 1% of the total data, this leads to a higher number of false positives (lower precision).

As the number of negative samples provided during training increased, we noticed recall dropped while the precision rose. In a sense, we are confusing the model. That is, the model is only making predictions when it is highly confident at the expense of its recall performance. The balance we are trying to strike is getting the model to predict roughly the same number of hidden positives that we expect to find. The graph in Figure 3.2 illustrates how our model performs

when asked to provide an increasing number of predictions.

To compare the positive and unlabeled learning results to one-class and multi-class classification, we again calculate f1-scores for each application. The average f1-score for PU learning with 1,000 runs per application is 0.6497, with an average precision of 0.6675 and an average recall of 0.6486. Table 3.5 shows a summary of the average f1-scores across all three techniques and Figure 3.1 depicts the distributions. Individual application f1-scores can be seen in Table 3.4 alongside the results from multi-class and one-class classification.

The average f1-score, precision, and recall for all applications and for the Signal application specifically, decline gradually with decreasing information across all three classification techniques used in this chapter.

Table 3.5 The average f1-score, precision, and recall for all applications combined and for the Signal application specifically, across all three techniques used in this chapter for classification.

	All Applications Combined			Signal Application Only		
	f1-score	precision	recall	f1-score	precision	recall
Multi-class Classification	0.7791	0.7839	0.7774	0.9685	0.9915	0.9466
PU Learning	0.6497	0.6675	0.6486	0.8708	0.9687	0.7913
One-class Classification	0.3290	0.2579	0.5631	0.7410	0.9146	0.6236

We note that using PU learning to classify our Signal application provides the second-highest average f1-score of 0.8708 with an average precision of 0.9687 and an average recall of 0.7913. In other words, our PU learning model was able to detect 79.13% of the hidden Signal samples, and when it claimed a sample belonged to Signal, it was right 96.87% of the time. We then tested the Signal application with 90%, 95%, and ultimately 99% of its samples hidden. Figure 3.3 shows the results of our model for the Signal application with varying amounts of hidden positive samples. In this graph, we note that accuracy decreases with an increase in the number of samples hidden from the model. Intuitively this result makes sense, e.g., we note the model only has 14 positive samples for training in the most extreme case (when we hide 99% of the Signal application samples). Despite this small number of positive training samples, our model's top 1,311 ($1,325$ total positives $-$ 14 training samples) predictions are 49% accurate.

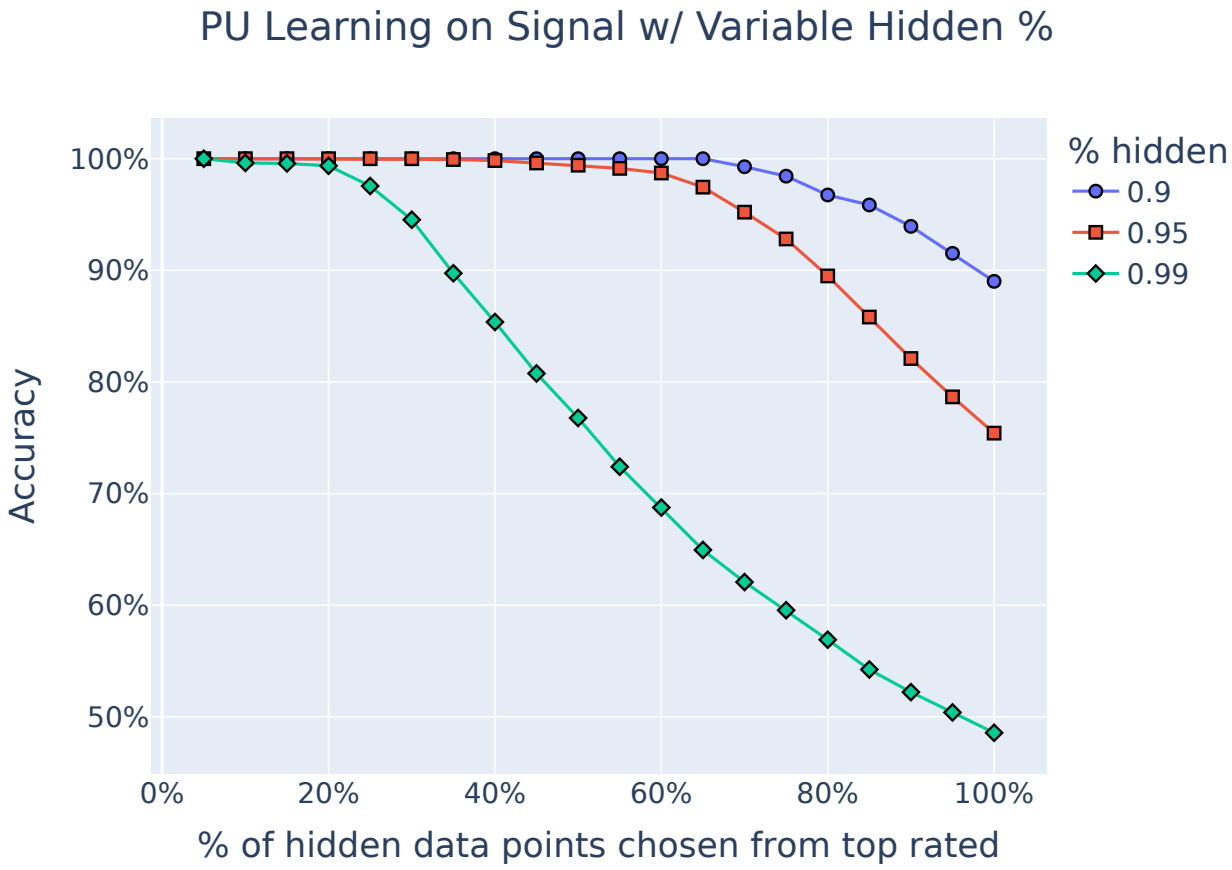


Figure 3.3 Accuracy of our PU bagging technique on the Signal messaging application, to illustrate the impact on performance as the number of points hidden is varied. The x-axis is scaled because the number of points hidden in the unlabeled set changes based on the hidden percentage. We note the variance of each data point is very low and, thus, is not included.

3.7 A closer look at the Signal application

Figure 3.1 depicts the distribution of the individual application scores combined across the three methods investigated in this chapter. To better understand how the Signal application performs, this section focuses solely on the distribution of Signal application f1-scores for each method.

As mentioned previously, we ran each experiment (multi-class classification, one-class classification, and PU learning) 1,000 times with a new random segment of our data used for training and testing in each iteration. Cross-validation helps ensure that one slice of the dataset doesn't bias our results. The resultant distribution of 1,000 f1-scores per method illustrates the

performance of the Signal application well.

Figure 3.4 shows the distribution of these 1,000 trials per method. As we hypothesized, and as Figure 3.4 suggests, multi-class classification achieves the highest performance ($\overline{f1} = 0.9685, \sigma = 0.0063$), positive and unlabeled learning the second best ($\overline{f1} = 0.8708, \sigma = 0.0157$), and one-class classification has the poorest performance ($\overline{f1} = 0.7410, \sigma = 0.0203$). Figure 3.4 shows that using the unlabeled data provides classification performance superior to that of one-class classification for the Signal application in our dataset. In other words, it is better to utilize the information contained in the unlabeled data when training with a labeled dataset comprised of a single class.

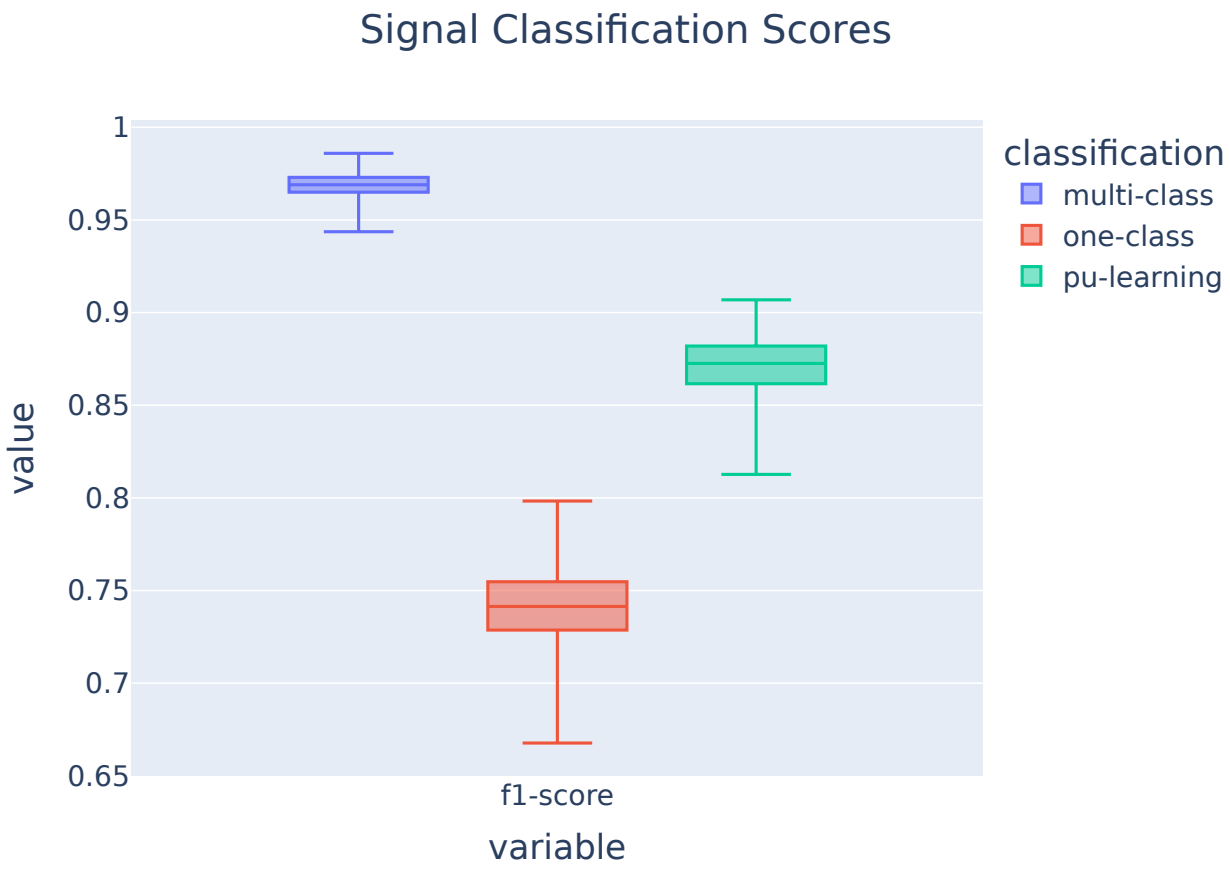


Figure 3.4 Distributions of 1,000 f1-scores for the Signal application for each of our three techniques for classification. The median values for multi-class classification, one-class classification, and PU learning are $0.9690, 0.7414,$ and 0.8726, respectively.

3.8 Discussion

This chapter investigates the effectiveness of positive and unlabeled learning for mobile network traffic classification. Specifically, we were intrigued by the privacy impacts of detecting when encrypted messaging applications were present on a network. To set a baseline of what one could expect in performance, we started with all of our data in a fully labeled state and executed a standard multi-class classification. To replicate a more real-world scenario, where access to fully labeled data is limited, we then discarded the labels belonging to every class but one. As we gave each application a turn at being this one, positively labeled class, we performed one-class classification using the OneClassSVM from `scikit-learn` and PU learning with bagging using an XGBoost classifier.

In one-class classification, we provided no negative training examples and, thus, expected the performance to be poor. With PU learning, we leverage the information contained within the unlabeled data set. As hypothesized, more information led to a better result. While not as good as the fully-labeled multi-class classification approach, the PU learning technique achieved double the average f1-score for all applications in the one-class experiments. There is one main caveat to this technique: it is difficult to tune the bagging approach without some estimation on the number of positive samples hidden in the unlabeled set.

The prediction threshold and the number of negative samples provided during each PU learning training round significantly impacts results. For our dataset, where each class represented approximately $1.5 - 10\%$ of the total samples, we found that a negative sample rate between $1 - 4\%$ performed best. While there was no linear relationship obvious, we believe there is some underlying relationship between the prevalence of the positive class and the appropriate number of negative training samples.

As we increased the number of negative training samples, models became more firmly convinced of their positive predictions but predicted fewer overall positives. In other words, more negative samples seemed to confuse the model and lowered its recall; however, when the model was highly confident a sample belonged to the positive class, it was almost always correct. Ultimately, we observed the highest performance when our classifier predicted approximately the same number of positive samples as the number that existed in the unlabeled dataset. Hence,

again, knowing the expected distribution of positives in the unlabeled dataset is essential for tuning the model's parameters. We also note that our prediction threshold was kept at 0.5, though researchers could also tune this to affect the number of positives predicted by a model.

Finally, in all three techniques presented in this chapter, we note that the Signal messaging application was highly susceptible to classification, and earned the second-highest f1-scores with PU learning of the applications we evaluated. More work is needed to determine if the entire genre of encrypted messaging applications is equally classifiable[21]. We suspect that the nature of encrypted messaging application usage is unique in the mobile application landscape, giving it a unique network traffic flow. This uniqueness could prove to make all encrypted messaging application susceptible to similar techniques as those presented in this study. In summary, training a model with a small labeled dataset containing only positive instances of the Signal application and finding the vast majority of hidden samples in an unknown dataset is possible and has, of course, significant implications for user privacy.

3.9 Related Work

3.9.1 Datasets

The research team in [6] created the dataset that inspired much of this work. We find their approach to labeling network traffic flows generated on Android applications to be the most accurate we have encountered in all of the literature. A similar technique of "de-multiplexing" flows generated by an application from that of other background processes or apps is described in [31]. It remains unclear how easy it would be to root a wide variety of Android devices in the future, perhaps making this approach less practical. Additionally, we do not believe this technique is feasible with iOS devices without root-level access to iOS (obtained through a process known as *jail-breaking*). Even then, we have found no examples of `strace` or equivalent being readily available on these devices.

Of note, we believe the precision of the timing features of this dataset could be improved if preserved in integer format instead of cast directly to a float from epoch time format. Additionally, we found minor discrepancies with the calculation of inter-arrival time statistics for the bi-directional flow portion in the MIRAGE-19 dataset. As we did not use inter-arrival timing

[21] A question we investigate in the next chapter.

in our work, the bi-directional flow calculations do not affect our results.

3.9.2 Traffic Classification Broadly

Many papers exist on network traffic classification, and several excellent surveys have been written, e.g., [10–12, 32, 33]. A handful of examples of network traffic classification are [13, 34, 35]. These papers all examine the ability to classify types of traffic from network data; however, there is a great deal of nuance in the methodologies and the conclusions presented. In short, there are many approaches to feature selection and techniques used in traffic classification, not all of which are machine learning (as in our study).

3.9.3 Mobile Application Traffic Classification

Closely related to our work in this chapter is [22] and [31], which both focus on mobile application traffic specifically. In [31], the authors present a framework they call AppScanner, that captures network traffic flows, processes them like ours, and performs various classification tasks on all of the applications. Notably, the traffic flows are generated synthetically rather than by human users, leading to repetitive traffic patterns. Synthetic traffic flows may impair the ability to generalize to real-world traffic flows, though the benefit of scalability is undeniable. The authors employ three broad approaches for traffic classification: multi-class classification using a classifier per flow length with a vector of packet sizes as the features, multi-class classification using a single large classifier with statistical features of the flows, and binary classification using a single classifier per application.

The multi-class classification technique using packet sizes for each possible flow length is a unique approach that performed well in their study. The single classifier approach is similar to the one we implement in Section 3.4. Their study does not explicitly state if the samples belonging to all other applications are negative or unlabeled, but our best interpretation is that they serve as negative examples to the binary classifiers. The binary classification technique is again similar to our work in that it is the first logical step in training a model on just one single application. We extend the approach in [31] by attempting one-class classification and PU learning.

In [22], this same research team extended the AppScanner framework, explicitly addressing the weakness in handling ambiguous traffic flows. They also demonstrate the intuition that

updates to applications can degrade the ability of a model to classify those applications several months after being trained.

In a very recent paper [36], the researchers build a framework that is ostensibly a direct upgrade to that of [31]. Their study also creates a framework for generating, capturing, and analyzing network traffic from Android applications. Their work attempts to identify both the application and action within that generated the traffic. Similar to [6], and by extension ours, this study uses low-level Linux logging to capture system calls on Android phones. We believe this is among the most comprehensive techniques for de-multiplexing packet captures from phones, guaranteeing the retention of only the target application's network traffic. Like many other studies where researchers generate their own datasets, the authors of [36] leverage tools to create much of the traffic synthetically. As noted previously, synthetic data generation is different from our approach and, while it does provide the ability to generate *much more data*, it might not provide realistic variance in network flow characteristics. Nevertheless, this work is comprehensive and provides a thorough approach to building a capture and classification framework for mobile applications.

More specifically than looking at just mobile applications, the researchers in [21, 23] all examine instant messaging applications. Notably, [23] targets iMessage on Apple iOS devices (in addition to OSX), which is a rarity in the mobile application classification literature. We suspect the lack of iMessage-related work is mainly due to the difficulty in rooting (or *jail-breaking*) Apple devices. The researchers present a generic attack presumably suitable for all messaging applications in their work. Their focus is on iMessage, and specifically on the actions taken by users in the application, though scripts drive their captures instead of human users. In addition, the research team also expands their work to assess the ability to predict the selected language within the application. Overall their results are impressive, achieving f1-scores above 99% for nearly every activity and very high accuracy for language detection after observing merely tens of packets.

The researchers in [21] present a framework for classifying the usage type in messaging applications. They adopt a unique approach to feature selection in that they describe network traffic as traffic-flows, sessions, and dialog, each of which has an increasing scope and amount of information. They provide one of the best descriptions of the feature selection process in the

literature. Their work focuses on WhatsApp and WeChat, two other prevalent encrypted messaging applications. While WhatsApp does use the Signal protocol for encryption, we find the Signal messaging application studied in our work to be more privacy-focused overall. They show great accuracy results for both of these applications and note that the random forest classifiers performed exceedingly well in their studies. Our results confirm that tree-based classifiers perform well in network traffic classification.

3.9.4 Learning with Unlabeled Data

PU learning and one-class classification are learning methods applied to a broad selection of domains, not just network traffic classification. In [16], the authors do an excellent job of summarizing the PU learning field and available techniques, some of which we applied in this study. In [37], a widely cited and highly regarded work in the PU learning space, the authors demonstrate the application of PU learning with the Kyoto dataset for anomaly detection in network data; this work is very closely related to what we do in this study, though their method is a two-step method that leverages reliable negatives.

Similar works to ours are presented in [18] and [38], two publications very closely related to one another. In these two publications, the authors leverage an extensive unlabeled dataset captured from a mobile network provider to do PU learning. Notably, the features used in these works contain detailed information about the user and handset generating the traffic. In our study, we limit ourselves to just the coarse data features observable from the network flows, i.e., we do not capture any information about the users or handsets. The works in [18] and [38] also leverage commercial traffic monitors embedded in the mobile network to provide a type of ground truth labeling.

In [19] and [30], researchers use one-class (or single-class as notated in [19]) SVMs to perform network traffic classification, just as we do in our one-class classification. The authors of [19] show an ability to correctly label many applications $90+\%$ of the time, but note less efficacy with certain datasets. Notably, this research team also uses a different feature set to represent network traffic flows than our work. In their classification, models are fed feature vectors of a fixed length where each feature is a single packet's size, modified by some constant based on the direction the packet was traveling.

The team in [30] leverages a compilation of one-class classifiers against network traffic to detect known and unknown applications. Their framework trains a single one-class classifier per application and compares it to a one-class support vector machine trained on all of the 'normal' traffic, containing many classes. They note the efficacy of this ensemble approach over just a single one-class support vector machine, which is not surprising considering the results of our work. As the authors note, including many classes in a one-class support vector machine's training will make it difficult for the classifier to distinguish between target and non-target traffic.

For an excellent summary of PU learning with code examples on a toy dataset, see [39]. The techniques and code snippets presented in [39] were quite useful in our early experimentation.

3.10 Future Work

Positive and unlabeled learning is applicable for problems faced in the real world and has much potential for network traffic classification. Adding different features, namely those in vector format representing *per packet* sizing, will be an exciting area to explore. In fact, in Chapter 5, we propose representing our bi-flows without statistically descriptive features.

Another obvious direction to pursue in PU learning for network traffic classification is leveraging deep learning models as the underlying classifiers. There are many examples in the literature of successful application of deep learning for network traffic classification. Thus, combining the bagging or two-step approaches with a deep learning model for classification could prove quite effective. We explore deep learning for network traffic classification in Chapter 5.

Finally, we seek to expand our dataset to include many different encrypted messaging applications, allowing us to attempt classification on the entire *genre* of encrypted messaging applications. Training on a genre of applications instead of a single application holds promise for identifying so-called "zero-day" observations of new applications in that genre. That is, can a model trained on a suite of encrypted messaging applications detect a never-before-seen application belonging to the same genre? A fundamental assumption for this research direction is that most of these applications will have similar network traffic patterns and thus similar fingerprints. We explore this question in Chapter 4.

3.11 Acknowledgements

A special thank you to Ethan Taylor for his assistance in capturing data, both building the infrastructure and working with the Android phones, during a summer internship as an undergraduate researcher. A great deal of gratitude also goes to Andrew Reed for the countless discussions on techniques, applications, and overall serving as a sounding board for this study.

CHAPTER 4
GENERALIZING MACHINE LEARNING MODELS FOR ZERO-DAY ENCRYPTED MESSAGING APPLICATIONS

4.1 Introduction

Encrypted messaging applications have seen a rise in usage over the past few years. These applications implement end-to-end encryption, so only the intended recipient can see the message's content. Users turn to these applications seeking more privacy in their conversations, yet information leaked through side channels can still help observers glean information about the user. The mere presence of these encrypted messaging applications on a network or the association with a user or IP address can reveal information that the user likely would rather not have exposed. In this work, we attempt to detect these encrypted messaging applications on a network using network traffic classification, specifically when no training data on the application being detected exists.

As discussed in Chapter 3.1, mobile network traffic classification is the subfield of network traffic classification focused on classifying mobile applications based on their representative traffic objects (i.e., packets or "flows"). While historically well-known ports and protocols were used for classification purposes, recent research has turned towards machine learning (ML) for this task. Much of the work done in this space leverages fully-labeled datasets, which are very difficult to come by in real-world applications. The sheer number of encrypted messaging applications makes it challenging to generate or acquire enough up-to-date labeled samples of network traffic flows from all applications in use. Furthermore, the frequent updates typical for mobile applications could quite possibly invalidate all of the labeled samples that one could acquire. All of this results in the reality that access to unlabeled data is ubiquitous, but access to labeled data is limited, a perfect scenario for positive and unlabeled learning.

Traffic classification can be used to answer questions about user behavior on the network. For instance, given a model trained on a specific encrypted messaging application, one could attempt to answer the question "Is anyone using the WhatsApp messaging application on the network?" On the other hand, traffic classification could also be used to answer a broader question, such as

"are users on this network communicating via any encrypted messaging application?" The difference is subtle. We ask a narrower question about one specific application in the former example. In the latter example, we are more concerned with general user behavior and detecting applications belonging to the broader genre. Notably, not every network traffic flow from a device needs to be accurately labeled, as even identifying just a single flow belonging to a target application indicates that the device is using the application.

This latter question, where we are interested in the presence of applications belonging to a genre, is problematic as mobile (including encrypted messaging) applications are constantly changing. New applications are frequently released, so machine learning models trained for this task will likely encounter never-before-seen applications, i.e., "zero-day" applications. Furthermore, existing applications are subject to continual updates, which could cause an application's features to drift so significantly that the application is essentially a zero-day application.

In this chapter, we first seek to establish how well positive and unlabeled machine learning methods from Chapter 3 generalize to the detection of zero-day applications in an unlabeled network dataset. In other words, because it is challenging to know *a priori* what encrypted messaging applications might exist on a network, can we create a class comprised of a handful of the most popular that *represent the genre* of encrypted messaging applications in order to detect other examples of this genre that are absent in training? Enabling the detection of previously unseen applications alleviates the real-world challenge of acquiring enough labeled data for the myriad of encrypted messaging applications.

One concern with this class representing a genre is whether or not the data can become 'stale' over time as more and more updates are released. Thus, we also investigate to what extent several months' worth of updates has on the effectiveness of models trained on a previous version of an encrypted messaging application.

To that end, the main contributions of this chapter are:

- Showing that positive and unlabeled machine learning models trained on statistical features from several applications from the encrypted messaging application genre can generalize for zero-day applications.

- Demonstrating that models trained with network traffic data from previous versions of the Signal encrypted messaging application effectively detect newer versions (i.e., the features do not necessarily drift significantly enough from version to version to invalidate previously gathered and labeled data.)

4.2 Background

4.2.1 Data

In this chapter, we combine the MIRAGE-19 dataset [6] introduced in Chapter 2 and discussed further in Chapter 3 with custom-generated traffic flows from six encrypted messaging applications. Manually generating mobile network traffic takes a long time, and we lessen that time requirement by using a publicly available dataset. Unfortunately, the MIRAGE-19 dataset does not contain enough examples of encrypted messaging applications, so we reverse-engineered the capture system from [6], and generated network traffic flows in the same manner, but for six encrypted messaging applications: BiP[22], Signal[23], Telegram[24], Viber[25], WhatsApp[26], and WickrMe[27]. Our reverse-engineered capture system, detailed in Figure 2.1, precisely labels network traffic flows from our encrypted messaging applications without concern that other network services or applications are included. The encrypted messaging applications were chosen based on their worldwide popularity and the geographical dispersion of their respective user bases. More details are in Table 4.1. For each of our six encrypted messaging applications, we generated 1,000 labeled network traffic flows. Combining these encrypted messaging application flows with those classes from MIRAGE-19 that have more than 1,000 samples, we have approximately 103,000 network traffic flows to conduct our experiments.

Network traffic flows for the Signal messaging application were created for the work in Chapter 3. Therein, we created 1,000 new samples of the Signal messaging application during the data generation phase of this project. That is, we have five unique encrypted messaging applications (BiP, Telegram, Viber, WhatsApp, and WickrMe) with 1,000 samples of network traffic flows in addition to our Signal messaging application which has 1,000 *new samples* from

[22]https://bip.com/en/
[23]https://signal.org/en/
[24]https://telegram.org
[25]https://www.viber.com/en/
[26]https://www.whatsapp.com/
[27]https://wickr.com/me/

version 5.43.7 as well as 1,000 samples used in Chapter 3 from version 5.25.7. These two Signal version releases were approximately nine months apart from one another[28] and the samples from the earlier version will be used to determine if the cumulative updates in those nine months are significant enough to essentially be considered a new application. In other words, we use the earlier version of Signal as training data to detect the newer version of Signal, gleaning insight into how often one might expect to have to retrain a model with newly labeled data (a significant time investment).

Table 4.1 Details regarding the six encrypted messaging applications used in this chapter.

Application	Notes
BiP	BiP is an encrypted messaging application operated by Turkcell, a Turkish mobile phone operator. BiP saw increased usage in early 2021 following controversial policy changes in the WhatsApp application.
Signal	Despite generating traffic flows for this application in our work described in Chapter 3, we have updated the application and generated traffic flows with the new version. Having two versions of this data allows us to explore if and how this particular application's network behavior has changed over time. Specifically, we investigate whether a model trained with network flows from version 5.25.7 can classify traffic generated from version 5.43.7, which is after approximately nine months of updates.
Telegram	Outside of Signal, Telegram is one of the most widely known encrypted messaging applications for privacy-minded users.
Viber	Viber is an encrypted messaging application owned by Rakuten, a Japanese electronic commerce company.
WhatsApp	WhatsApp is a universal, end-to-end encrypted messaging application owned by Meta (formerly known as Facebook).
WickrMe	WickrMe is an encrypted messaging service acquired by Amazon Web Services in 2021. Wickr Me is a product aimed at personal use. At the same time, Wickr provides an enterprise messaging service and a product known as RAM (which provides secure messaging capabilities to the United States Department of Defense.)

[28] https://github.com/signalapp/Signal-Android/commits/main

4.2.2 Positive and Unlabeled (PU) Learning

As in Chapter 3, we utilize PU learning for this study. PU learning, a form of semi-supervised learning, allows for training models with only a small portion of labeled data combined with a typically larger set of unlabeled data. This constraint of having just a small percent of known positive labels for data models is widespread in real-world situations where it is far easier to capture unlabeled data than labeled data. The MIRAGE-19 dataset is our 'unlabeled data' throughout this study during training and classification tasks. The MIRAGE-19 dataset has associated labels, but we withhold these labels from our models during training. In other words, while our models will be given negative training samples in this chapter, these are selected from our unlabeled data and are not true 'known negatives.' We retain the labeling strictly for evaluation purposes (i.e., when our models make classifications on the unlabeled data as to whether or not a sample is an encrypted messaging application, we use the original labeling to evaluate those predictions).

We implement both common techniques for PU learning: bootstrap aggregation (known as 'bagging') and 'two-step.' In both approaches, we have a small set of positively labeled data, our 1,000 flows per encrypted messaging application, and a large set of unlabeled data, the approximately 100,000 flows from the MIRAGE-19 dataset. During training, true positive samples are always provided to our models. In contrast, the negative samples are either random guesses, as in the bagging approach, or 'likely negatives' as in the two-step technique. In either algorithm, the negative training examples provided during training are not truly 'known negatives' but rather random or educated guesses from the unlabeled samples.

4.2.3 Zero-Day Applications

Throughout this work, we use the term "zero-day" encrypted messaging applications to describe an encrypted messaging application not seen by a model during training. We first referenced this term in Section 4.1 and expand on it here for clarity. This concept is meant to mimic the real-world phenomenon where, for many reasons, models will be making classification decisions on samples belonging to classes not seen during training.

For example, one might need to detect the presence of encrypted messaging applications on a given network or post-mortem in a collection of network traffic flows. It is not feasible that

up-to-date, labeled network flows will be available for every application that might appear in the unlabeled network traffic. Generating and labeling accurate network traffic data for mobile applications is a time-consuming task with many architectural challenges in the demultiplexing of flows belonging to the target application from the other applications and services running on the mobile device.

A practical approach to handling this constraint would be to use labeled data for a subset of encrypted messaging applications representative of the genre during training. The individual applications might be selected based on the *most likely to be found* or, as we do in this study, the *most popular* examples of the genre. We discuss our approach to this constraint in more detail in Section 4.3.

4.3 Overview

In this section, we provide an overview of our approach to the problem of detecting zero-day encrypted messaging applications in an unlabeled dataset. We first describe the features utilized during training. Following that, we present the design of the experiments conducted in this study.

4.3.1 Feature Selection

In its most raw state, network traffic data is captured by the individual TCP/IP packet. Each packet belongs to a conversation between two hosts called a "flow." Each packet contains header information and application data. Because the application data is a large portion of a given packet and is usually encrypted, discarding this payload and storing only the headers from each packet in a flow makes sense. This results in, namely, significant storage requirement savings. From these headers, we can create summary statistics describing the flow. Similar to the process and features described in Sections 2.4 and 3.3, we process the packet captures from the encrypted messaging applications detailed in Section 4.2.1 into our data format, which is identical to that of the MIRAGE-19 data we use as our unlabeled dataset.

The 5-tuple of source IP address and port, destination IP address and port, and transport layer protocol uniquely identifies each flow. This 5-tuple serves as the key for each record in our dataset. Each flow contains three primary sets of descriptive features: metadata, packet length data, and inter-arrival timing data. Metadata is overall summary statistics about the flow, e.g.,

the total number of bytes sent back and forth or the total duration of the flow. Packet length and inter-arrival timing data statistics describe the flow in the upstream, downstream, and bi-directional directions. Table 3.1 and Table 3.2 list the features for the MIRAGE-19 data and our encrypted messaging application traffic.

In Chapter 2 and other work done in this field (e.g., [19]), timing features are sometimes not used during training and classification tasks. Typically these features are removed because, in those specific works, there is not a substantial impact on classification performance, and reducing the total number of features in a dataset can yield a significant improvement in training times. In Chapter 2, specifically, we also remove timing features as there is the possibility that, because our encrypted messaging application data is being combined with data from a different capture system, differences in the network media could result in timing biases that skew our results. For this chapter, timing features remain in the dataset as generalizing to zero-day applications appears to lean more heavily on timing than other work that uses fully-labeled data. Specifically, we noted a significant drop in performance when we removed timing features. We further discuss timing features and their importance in Section 4.4.

To reduce potential bias in our timing features, but still allow for some aspects of timing to be used by our models in this chapter, we considered which timing features to discard. Given that our network traffic flows contain inter-arrival timing features for packets leaving our network (upstream), packets entering our network (downstream), and the interleaving of the two (bi-directional), we choose to discard all timing in the upstream direction (which includes bi-directional traffic as that contains upstream traffic). These upstream timing features are most likely to carry bias from our local capture system's physical properties because the only influence on recorded inter-arrival times in the upstream direction is the Wi-Fi spectrum between our Android phones and the laptop serving as the access point. Figure 2.1 illustrates our capture system for visual reference. The downstream data will carry minimal bias as the recorded timestamps are at the access point, demarcating our local capture system from the wider internet. The application server's behavior and the packet's path through the internet's architecture will largely drive this downstream data's inter-arrival timing characteristics.

4.3.2 Experimental Design

Our work in this chapter focuses on detecting zero-day applications belonging to the genre of encrypted messaging applications. Figure 4.1 visually depicts this process with key points indicated by numbered stars. We begin with 1,000 labeled samples from each of the six encrypted messaging applications listed in Table 4.1 (key point #1). One of these six encrypted messaging applications is removed at a time (e.g., Signal in Figure 4.1), which serves as the 'zero-day' application for that training iteration (key point #2). The chosen 'zero-day' application has its labels removed and the remaining five applications are combined into a single class representing the genre of encrypted messaging applications (key point #3). The now unlabeled 'zero-day' application is hidden among the rest of our unlabeled data from the MIRAGE-19 dataset (key point #4). The bagging and two-step algorithms are implemented (key point #5), with the combined class of encrypted messaging applications serving as known positives. Negative training samples are provided to the models based on the particular algorithm, i.e., bagging selects random negative samples with replacement and the two-step algorithm first attempts to guess the *most likely* negatives. This entire process is repeated six times (key point #6) until each of the six applications has been used as the 'zero-day' application.

Once the combined class is created and the current iteration's zero-day application is unlabeled (i.e., hidden amongst the unlabeled data set), we begin our training. In this study, we use the bagging and two-step approaches to PU learning as discussed in Section 3.6. In Section 3.6, we implemented PU learning using the bagging technique with XGBoost as the underlying classifier. In that work, we chose bagging because it was consistently the most accurate technique, though the two-step approach described in that section was competitive for many of the applications. Because we added new applications to the dataset, we also include the two-step approach in this work for evaluation. Section 4.4 discusses parameter tuning and other implementation specifics.

Finally, the last type of zero-day application we consider in this study is the 'updated application' type, i.e., an app that has had significant enough feature-distribution drift that models trained with previous versions may no longer effectively recognize the new version as the same class. In this study, we use the Signal application to investigate whether an 'updated application' can be classified correctly. Signal, one of our six encrypted messaging applications,

was also used in our previous work from Chapter 3. Those flows from the previous work (version 5.25.7, released 28 October 2021) are viewed as our 'old data', while the traffic flows generated for this chapter's work (version 5.43.7, released 21 July 2022) are viewed as our 'new data.' In this case, the training data is the 'old data' from Signal, with the 'new data' from Signal hidden in the dataset. This experiment does not use other encrypted messaging applications listed in Table 4.1 for training purposes, though they remain hidden amongst the unlabeled data.

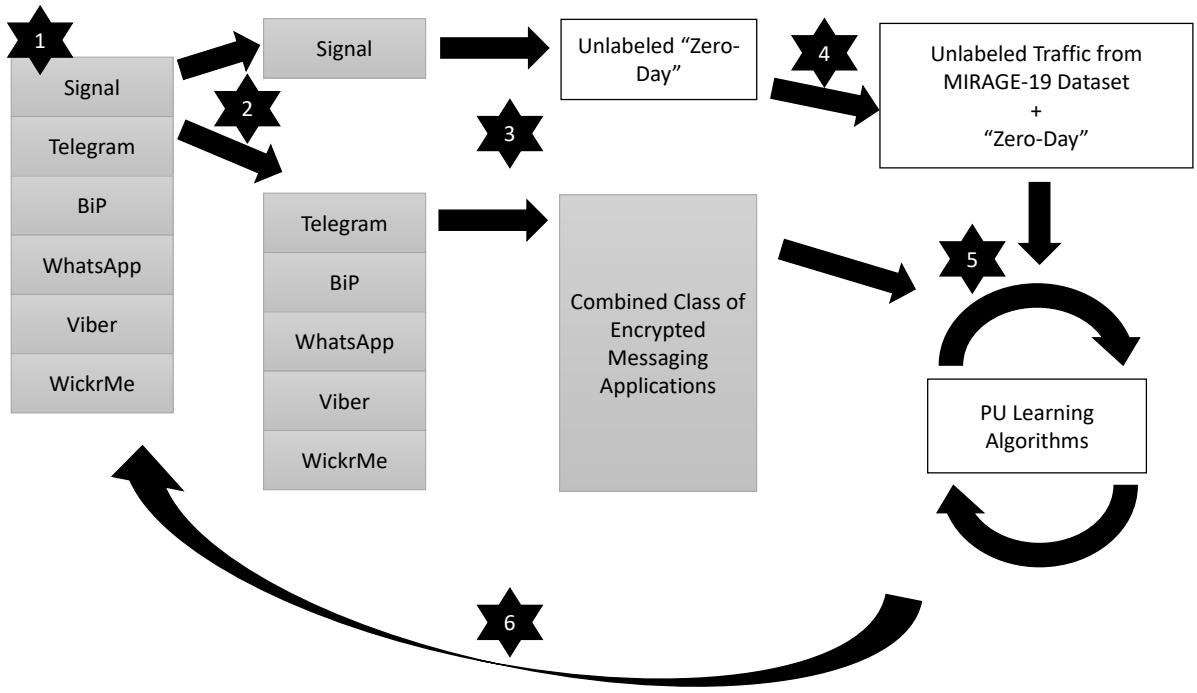


Figure 4.1 The process of creating a zero-day application is depicted in this figure (e.g., in this figure, we show the Signal messaging application being treated as our zero-day). Every encrypted messaging application has a turn having its labels removed, where it hides among the rest of our unlabeled data. The goal is to train a PU learning model with a combined class of the remaining labeled encrypted messaging applications. This combined class represents the genre of encrypted messaging and is used to determine how well models can generalize to zero-day applications.

4.4 Details and Results

4.4.1 Model Selections and Tuning

As discussed in the previous section, the two implementations for PU learning used in this study are the bootstrap aggregation (bagging) and two-step algorithms. In both approaches, the goal is to take advantage of the latent information in the unlabeled dataset combined with the

few positively labeled samples. We discuss our specific implementations of each algorithm in the following paragraphs.

Bootstrap aggregation for PU learning is an iterative process where a subset of unlabeled samples are randomly chosen with replacement (i.e., a sample can be chosen more than once) to temporarily treat the samples as negatives. Then, an underlying classification model is trained with these temporary negatives and all known positives. Finally, a prediction is made for every unlabeled sample that was not chosen (often called "out-of-bag"). After just a single round of training, we do not expect the results to be very good. However, over hundreds or thousands of iterations, the average of predictions that each unlabeled sample receives begins to coalesce into a much more accurate result. More details about bagging are given in Chapter 3.

In our bagging study, after experimenting with Random Forests, Support Vector Machines, and XGBoost models [25], we settle on XGBoost as our underlying classifier as the results are consistently better for our dataset and experiment parameters. For each of the three distinct experiments that follow in Sections 4.4.2, 4.4.3, and 4.4.4, we begin with parameter turning on our XGBoost classifiers. The parameters having the largest impact on performance, and thus the ones we tune, are the minimum sum of instance weights needed in each child node, the maximum depth of the decision tree, and the learning rate.

Next, we sought to determine the highest-performing negative sample size for each bagging iteration. That is, we investigated how the number of unlabeled samples randomly selected to be temporarily negative for each iteration affected performance. With our dataset and the XGBoost classifier, we found 100 negative training samples for each iteration to be the minimum required for decent performance. The classification performance could be increased by increasing the negative training samples up to 3,500. It should be noted, however, that more negative training samples slightly increases the training time for the XGBoost model, but not so significantly to deter us from the performance gains obtained. Figure 4.2 depicts the impact from increasing the number of negative training samples on our bagging performance.

With our parameters tuned and the number of negative training samples determined, the final step was to establish how many training iterations and predictions were required to attain good results. We begin by noting an unlabeled sample not chosen for a specific training iteration will receive a prediction once the XGBoost classifier is trained in that round. Thus, after a

hypothetical 100 iterations, each unlabeled sample will have received approximately 100 predictions. This value is *approximately* 100 as it will likely have been chosen at least once to serve as a negative, which means it will not receive a prediction that round. These 100 predictions are averaged into a single prediction probability of it belonging to the positive class. At some point, the linear time expense to keep making predictions does not significantly affect the average score an unlabeled data point receives. We found that a minimum of 100 training iterations was necessary to create a robust score, but more than that did not significantly change the results.

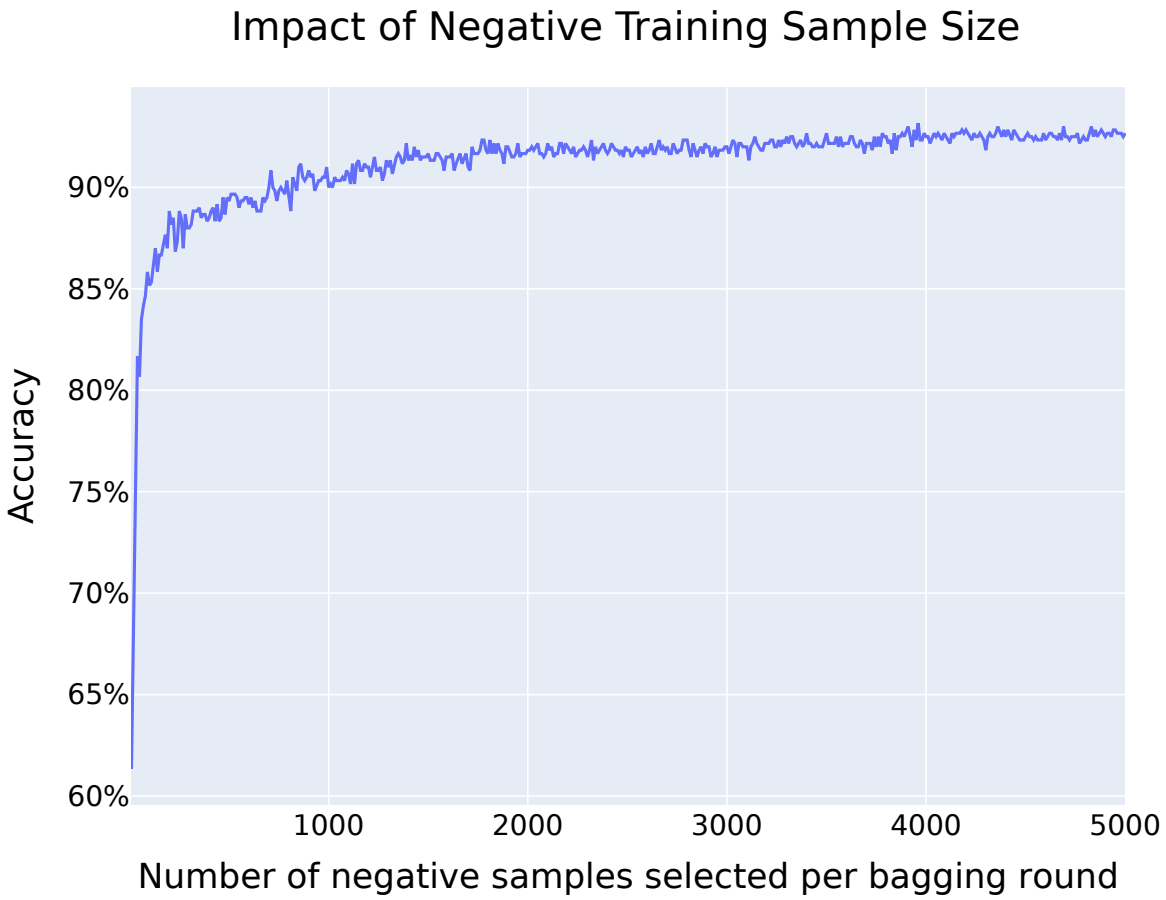


Figure 4.2 This figure illustrates the effect of changing the number of negative training samples shown to our model during bootstrap aggregation. As the number of negative training samples shown to our model increases, the accuracy obtained on classification also increases. On the lower end, at 50 negative samples, we achieve just over 81.2% accuracy. Increasing the number of negative training samples to 200 results in an improvement to 89.0% accuracy. Increasing the number of negative training samples to 5,000 results in a steady increase in accuracy, with 92.8% as the best accuracy for approximately 3,500 samples.

When implementing our two-step algorithm, we also use the XGBoost classifier with the same approach to parameter tuning. The difference from our bagging approach is mainly in identifying the negative training samples used in making predictions against unlabeled network traffic flows. For the two-step model, our first goal was to identify "reliable negatives," meaning samples that were so unlikely to be a part of our positive class that we assumed them to be negative.

Identifying these reliable negatives is step one of the two-step algorithm. There are many techniques for determining a good set of reliable negatives, and most are iterative in nature. Some studies utilize a clustering algorithm, with the reliable negatives selected from the cluster(s) that are furthest from the known positives. Other studies use a binary classification where every unlabeled sample is treated as if it were negative, and then each unlabeled sample is given an initial prediction. Suppose the initial prediction for a given unlabeled sample falls below a specific threshold (which researchers can set as they see fit). This sample is then changed from unlabeled to negative (i.e., it is now treated as "reliably negative"). In either case, known positives and reliable negatives are used for training in further iterations, with each yet unlabeled data point receiving predictions. If subsequent training rounds predict unlabeled samples to belong to our reliably negative class, they are added. Step one can be as aggressive or as conservative as needed. In the most exhaustive case, iteration continues until convergence is reached and no unlabeled samples receive predictions that fall below the specified threshold. Alternatively, step one can be stopped once the number of newly added negatives falls below a given target value (i.e., if a training iteration results in only ten new reliable negatives, we can simply stop the process as adding such a small number of negative training examples is likely not worth the cost of training time).

For our two-step implementation, we use a binary classification in step one of the two-step algorithm to first assign each of our unlabeled network traffic flows a probability of belonging to the positive class. In all of the experiments in Sections 4.4.2, 4.4.3, and 4.4.4, this initial prediction assigns several hundred unlabeled samples to the reliable negative class. We then iterate for 10 rounds using these new reliable negatives and our known positives, training and predicting against the remaining unlabeled data points. By the end of the 10th round, the number of new reliable negatives being added is less than 100 and we reach a sufficient stopping point.

Finally, after 10 rounds of step one where we identify these reliable negatives, we complete the two-step algorithm with a final prediction. The final set of reliable negatives are given to an XGBoost classifier along with our known positives for a prediction against all the remaining unlabeled points. These final predictions are then evaluated for accuracy and the algorithm is complete.

4.4.2 Detection of Control Samples

As our first experiment in this study, we do a classic train/test split with all our encrypted messaging applications. This first step aims to establish that a single class representing the encrypted messaging genre can, at a minimum, detect the applications that comprise this single, combined class (i.e., not zero-day applications). After determining what, in essence, is the peak performance we should expect, we will move on to the more challenging task of our models generalizing to zero-day applications.

For the design of this task, 10%, or 100 samples, from each of our encrypted messaging applications are unlabeled and hidden amongst the rest of the unlabeled data from the MIRAGE-19 dataset, totaling 600 target control samples. Next, we combine the remaining 90%, or 900 samples, from each encrypted messaging application into a single class representing the entire genre. This design represents the phenomenon where a model is trained with encrypted messaging applications representing the genre as a single class and then encounters traffic flows from the same applications in an unlabeled network traffic dataset. This experiment is repeated 1,000 times with different data splits for training and testing to ensure robust scoring.

We report results for our bagging and two-step approaches to PU learning after a few more fine-grained details on our process. In both techniques, we tune our model parameters to attain improvements over the default settings, though XGBoost models performed reasonably well with no tuning in our experiments. After tuning, the model is trained with the best parameter set and used to make final predictions against the unlabeled dataset regarding which network traffic flows belong to the combined class. Specifics about the model selection and parameter tuning are in Section 4.4.1

We expect the number of positive samples hidden among the unlabeled data to be relatively small compared to the genuinely negative samples. In other words, on a live network, the amount

of encrypted messaging application packets as a percentage of the total packets one should reasonably expect would be low. After the complete classification process, each sample in the unlabeled dataset has received a final predicted probability of belonging to the positive class. Typically, models predict a positive or negative sample based on a 50% cutoff score. If a sample in the unlabeled dataset has a .512 predicted probability, the model labels it as a '1.' When making classifications for a class that we expect to have very low representation (in this case, 600 samples out of more than 100,000), it is often helpful to set a higher cutoff score to achieve more confidence in the predictions and fewer predicted positives. A sample that receives a .999 predicted probability is generally more likely to belong to the positive class than one receiving a predictive probability of .512.

Thus, when evaluating our model's classification performance, we perform a 'Top-N' evaluation with a sliding N-value. In other words, if we know that we hid 600 control samples in the unlabeled dataset, we take the 600 predictions having received the highest prediction probabilities and evaluate how many of those 600 predictions were the actual hidden positives. Using bagging, our model correctly identifies 557 of the 600 hidden positives with its 600 best predictions. Or, in other words, selecting the 600 samples most likely to be positive yielded a 92.8% accuracy. When the two-step technique was implemented, the top 600 predictions contained 536 hidden positives, or an 89.3% accuracy, slightly lower than the bagging technique, but still quite high. Figure 4.3 shows these results.

4.4.3 Zero-Day Encrypted Messaging Application Detection

Now that we have established that our encrypted messaging applications are detectable in our unlabeled dataset (Section 4.4.2) we next examine the ability to detect these same encrypted messaging applications when they are *not a part of the positive class*. To achieve the effect of encountering a zero-day encrypted messaging application, we implement our bagging and two-step algorithms after completely removing one application before training. This removed encrypted messaging application has all its labels changed to that of the unlabeled dataset, effectively turning it into a zero-day, never before seen application. We repeat the experiment six times, allowing each encrypted messaging application a turn at being the zero-day application. Using each encrypted messaging application as the zero-day application provides intuition on how

individual applications being included or removed in the genre-representing combined class impacts the ability to generalize to zero-day applications. Figure 4.1 details this process.

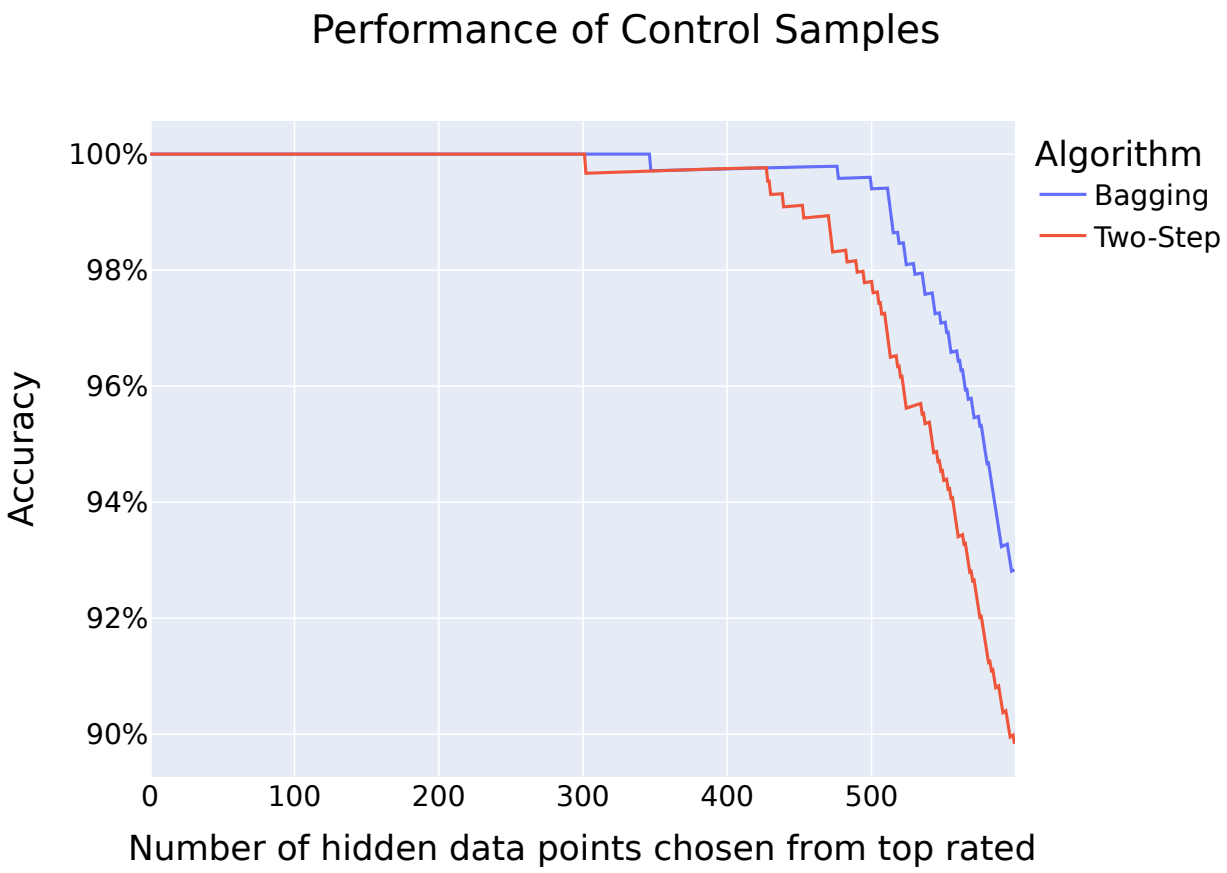


Figure 4.3 This graph shows the percentage of unlabeled traffic flows that were classified accurately from the hidden positive control samples from our six encrypted messaging applications. Of the top 600 flows in our unlabeled dataset, ranked by prediction probability, 557 are true hidden positives, providing a 92.8% accuracy for the bagging approach to PU learning. The two-step algorithm correctly identifies 536 out of 600, yielding an 89.3% accuracy.

Each encrypted messaging application in our dataset has 1,000 samples of network traffic flows. Thus, after removing the labels from our current zero-day application, each iteration of our algorithm has 5,000 positively labeled samples ($5\ \texttt{apps} \times 1,000\ \texttt{flows} = 5,000\ \texttt{samples}$) that we combine into our single class representing the genre. Conversely, the number of hidden samples is simply the 1,000 flows belonging to the current zero-day application. We do not simultaneously hide control samples from the five encrypted messaging applications as we demonstrate the ability

to detect them in Section 4.4.2.

As in Section 4.4.2, we utilize the XGBoost classifier as the underlying classification model for our bagging and two-step algorithms. We tune each classifier in the same fashion described in Section 4.4.1 to improve the classification performance over the default parameter settings. The tuning was not exhaustive, and it is reasonable to expect that slightly higher performance could be obtained with more extensive and expensive tuning. Lastly, we note that, during parameter tuning, we sought to optimize the *average performance of all applications combined* rather than any individual application. As discussed in the subsequent paragraphs, our model generalizes to some individual applications better than others, and various parameter settings affect each application's scores differently.

Figure 4.4 depicts the results of our zero-day detection experiment using the bagging algorithm. We present the result for each encrypted messaging application individually to illustrate how well we detect it as a zero-day application during its turn. We also present an average of the six applications to provide intuition into the overall ability to generalize to zero-day encrypted messaging applications. As mentioned in the previous paragraph, the average classification accuracy across all of the 'zero-day' applications is the metric we optimized for during parameter tuning. While we do not expect the average performance would change drastically, it is feasible that performance on a given application could improve had we selected specific parameter settings to optimize that one application. The results are interpreted similarly to those in Section 4.4.2. However, in this case, we select 1,000 samples with the highest prediction probabilities of belonging to the positive class, as that is how many samples from each encrypted messaging application we unlabeled during its turn at being our zero-day application.

With our bagging algorithm and chosen parameters, the encrypted messaging applications BiP and Viber were the most easily classifiable by our XGBoost model. Our model made the top 1,000 predictions for unlabeled samples that are likely to belong to the positive class (which includes the other five encrypted messaging applications). Out of these predictions, Viber accounted for 66.6% (equivalent to 666 predictions). When BiP was the zero-day encrypted messaging application, 67%, or 670, of the 1,000 top predictions belonged to BiP. Signal was the encrypted messaging application least susceptible to zero-day detection when using the other five applications combined as the positive class. Of the top 1,000 predictions made by the model

seeking Signal as a zero-day application, only 499, or 49.9%, were correct. The average across all six applications was approximately 59.1%.

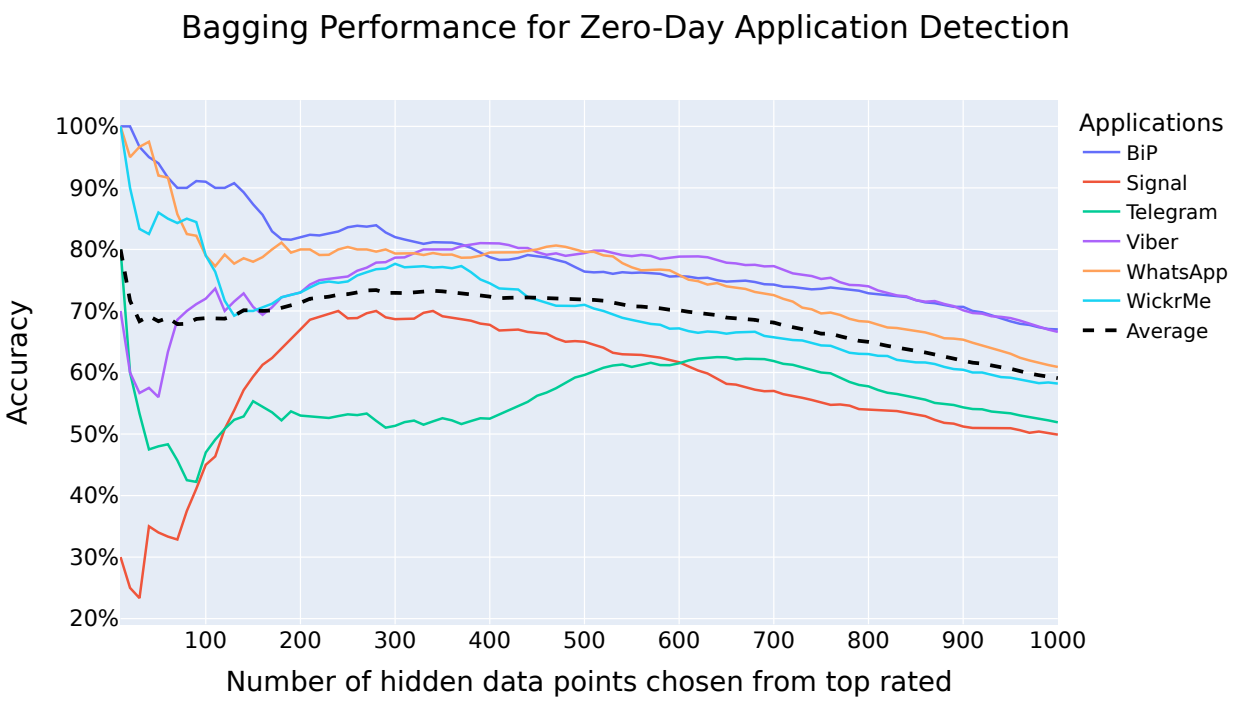


Figure 4.4 This graph shows classification accuracy for the bagging algorithm with parameters tuned to optimize the average accuracy score for the top 1,000 predictions. Telegram and Signal were the least susceptible to zero-day detection, whereas BiP and Viber were the most. Taking the top 1,000 predictions from our model for each application's turn at being the zero-day results in an average accuracy of 59.1%. The most susceptible application was BiP, barely surpassing Viber, with 67% of the hidden samples detected. The least susceptible application was Signal, with a classification accuracy of 49.9%; this result, while below 50%, still represents significant learning, i.e., these results are significantly better than random guessing.

During our parameter tuning, we observed that individual application performance varied with different parameter settings, suggesting there is no one best combination of parameter settings for the entire genre. The parameters we selected optimized the average of all applications. In contrast, if the goal was to optimize for a particular application, say Signal, we could likely achieve higher performance for that one specific application as a zero-day. When optimizing for one application in particular, it is unlikely that we would attain the highest performance across all applications on average. In other words, our average accuracy would likely decrease if we optimized for just one specific application. Our overall goal, however, is to detect

any application from the genre, not just one.

Figure 4.5 shows our two-step algorithm's zero-day detection results. The results are interpreted similarly to the bagging results shown in Figure 4.4. Again we note that Viber and BiP, though with greater separation this time, are the two most susceptible to zero-day detection in this experiment. When Viber is the zero-day encrypted messaging application, 61.4% of the top 1,000 predictions are classified accurately. Furthermore, again, we observe Signal and Telegram as the two encrypted messaging applications least susceptible to detection when chosen as the zero-day application. With the two-step algorithm, Signal only has a 46.1% accuracy for the top 1,000 predictions. The overall average across all six applications was approximately 53.3%. Given there are approximately 100,000 unlabeled samples, we note that this still represents a significant improvement over random guessing.

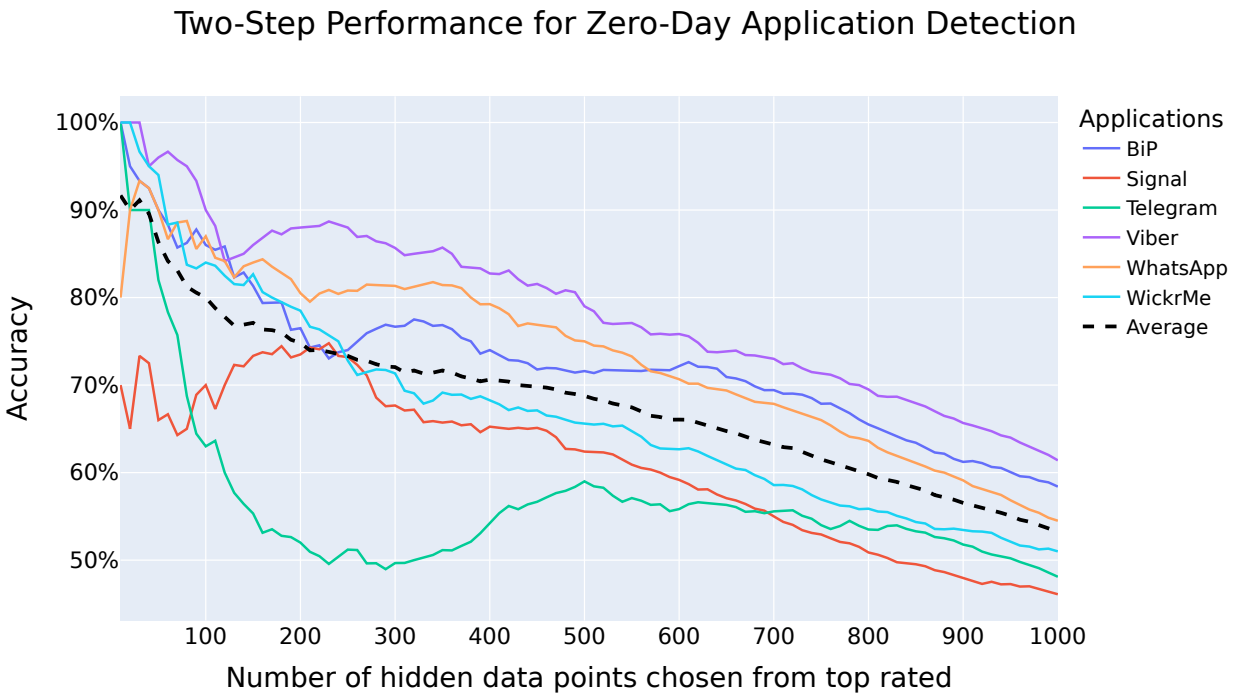


Figure 4.5 This graph shows classification accuracy for the two-step algorithm with parameters tuned to optimize the average accuracy score for the top 1,000 predictions. Taking the top 1,000 predictions from our model for each application's turn at being the zero-day results in an average accuracy of 53.3%. The most susceptible application was Viber, with a classification accuracy of 61.4%. This is a four percentage point margin over the next most susceptible application, BiP. The least susceptible application was Signal, with a classification accuracy of 46.1%.

We note that, similar to the control tests in Section 4.4.2, the two-step algorithm always performed worse than our bagging algorithm. Table 4.2 shows the overall results. While the zero-day detection of encrypted messaging applications is less accurate than classifying the control samples discussed in Section 4.4.2, we reiterate that no samples from the zero-day class are in the combined positive class given to our model in this section. That the models can classify an unseen application with some degree of success is significant and suggests that the genre of encrypted messaging applications shares certain characteristics that enable this classification. As noted in Section 4.2.1, there are over 100,000 samples in the unlabeled dataset. If we asked our model to randomly guess which of those 100,000 unlabeled samples were the zero-day application's 1,000 hidden samples, one would expect an accuracy on the order of 1%. Thus, to attain over 50% average accuracy with both algorithms in this study demonstrates that significant learning has occurred with our models.

Table 4.2 This table shows the overall results for zero-day encrypted messaging application detection. In every case, we observe a decline from the bagging accuracy, 59.1% on average, to the two-step accuracy, 53.3% on average.

Encrypted Messaging Application	Bagging Accuracy of Top-1000 Predictions	Two-Step Accuracy of Top-1000 Predictions
BiP	67.0%	58.4%
Signal	49.9%	46.1%
Telegram	51.9%	48.1%
Viber	66.6%	61.4%
WhatsApp	60.9%	54.5%
WickrMe	58.2%	51.0%
Average	59.1%	53.3%

4.4.4 Evaluation of Feature Distribution Drift in the Signal Messaging Application

As a final experiment in this chapter, we explore the idea that updates to mobile applications can lead to the phenomenon known as *feature distribution drift*. We take initial steps towards determining to what degree encrypted messaging application traffic flows from previous versions of an application can detect current versions. In this study, we use the Signal messaging application as we had previously collected more than 1,000 network traffic flows for the work

presented in Chapter 3. We used those "old" samples to train our models using the bagging and two-step algorithms to detect zero-day samples from the "new" version of Signal.

We again use the XGBoost classification model in this section for both algorithms. The models are tuned similarly to the tuning discussed in the previous two sections. In this case, we provide our models with 1,000 positively labeled samples from version 5.25.7 of the Signal encrypted messaging application. Amongst our unlabeled data set are 1,000 hidden samples from version 5.43.7 of Signal. We note that this newer version of Signal was released approximately nine months after version 5.25.7.

Results are presented similarly to the previous two sections, where we evaluate the accuracy of our models' sliding top-N predictions for both bagging and two-step PU learning algorithms. Figure 4.6 shows these results. Because we are looking for 1,000 samples of the newer version of Signal, we use that number as the maximum value of N. In the bagging algorithm, our XGBoost model trained with the older Signal samples correctly identified 881 of the hidden 1,000 samples belonging to the newer version, i.e., an accuracy of 88.1%. With the two-step algorithm, we obtained an accuracy of 76.8%, or 768 of the 1,000 samples correctly identified. As with the previous two sections, we again observe the two-step algorithm's performance lag compared to the bagging algorithm in classifying zero-day applications.

These results suggest that training data for encrypted messaging applications may retain usefulness in detecting newer versions of an application after months have passed. Compared with our classification study in Section 4.4.2, the bagging model in this section underperforms by about 4.7%. We note that, in this study, the other five (non-Signal) encrypted messaging applications are also hidden among the labeled data, making correctly identifying *only* the new Signal samples a more challenging task. In other words, we observe this model, which was trained on old Signal data, classify network flows from the new Signal data as well as some of the other encrypted messaging applications (consistent with the previous sub-section). The models herein detect many Signal encrypted messaging flows from the version nine months newer than what the model was trained on, indicating that the feature distribution drift was not so significant as to render those older training samples useless.

Before making broad conclusions, however, more work is needed. Even if many applications in the genre of encrypted messaging applications show similar resilience to feature distribution drift

after many months of application updates, a major update could change the network traffic patterns of the application significantly. Assuming this precedent holds, the main takeaway of this work is that organizations should continue to update datasets with current samples from encrypted messaging applications but that there may be some grace period where older data remains useful.

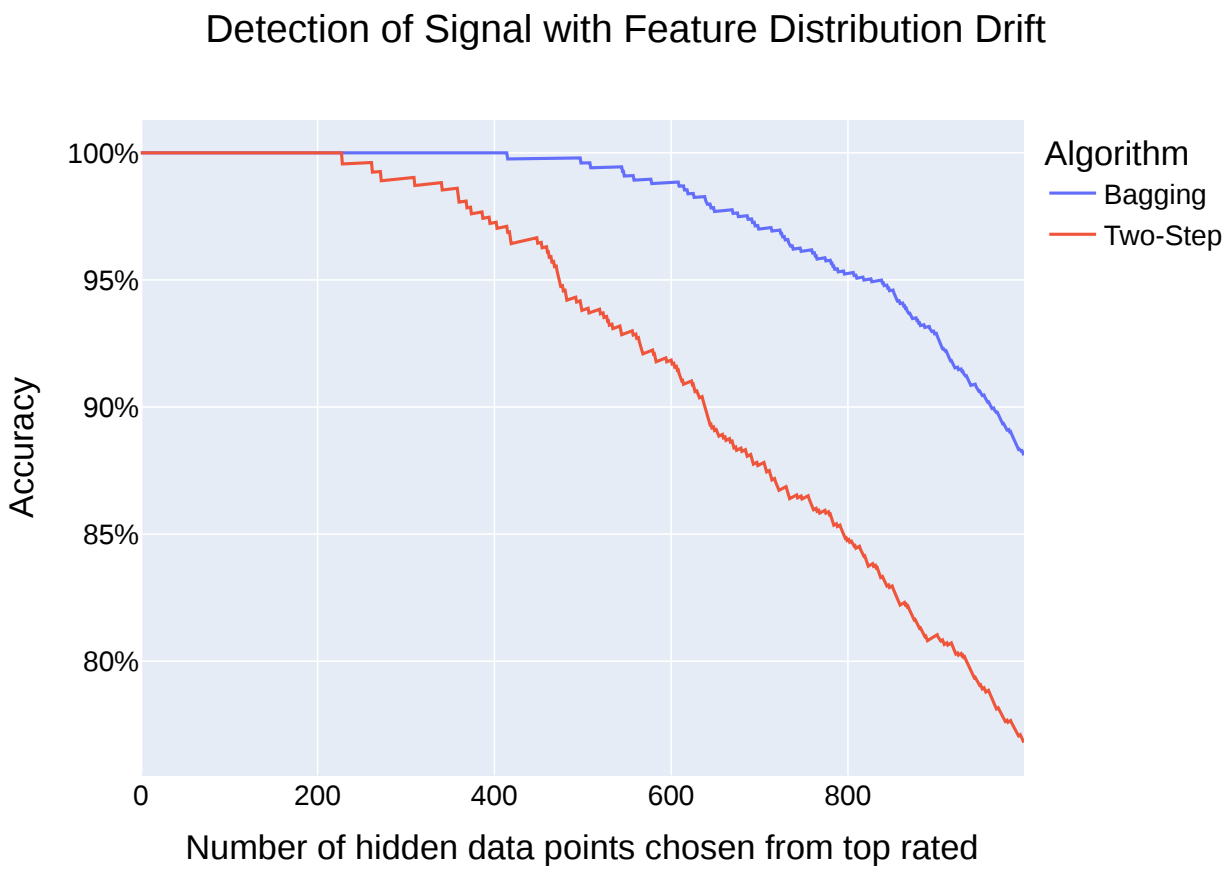


Figure 4.6 This graph shows classification accuracy for the 1,000 unlabeled samples receiving the highest prediction probability for belonging to the newer Signal (version 5.43.7) flows by our model trained on older Signal (version 5.25.7) flows from nine months prior. Our bagging and two-step algorithms correctly label 881 and 768, respectively, of their top 1,000 best guesses as to which of the more than 100,000 unlabeled flows belong to the newer version of the Signal encrypted messaging application.

4.5 Related Work

The dataset and collection method used for our work in this chapter is greatly inspired by [6], which shares many similarities to the techniques in [31]. In these works, the authors develop a highly accurate way of pulling packets belonging to a particular mobile application out of the larger TCP/IP traffic stream. These approaches to demultiplexing network traffic remain some of the most accurate and complete approaches we have seen in recent research.

Many techniques exist for network traffic classification in the published literature. These implementations can range from using classic approaches leveraging ports and protocols to traditional machine learning to, more recently, deep learning implementations. Several comprehensive surveys on network traffic classification can be read in [10–12, 32, 33].

As we do in this chapter and the previous one, learning with unlabeled data has recently gained much popularity. See [16] for a comprehensive survey. In [18, 38], the researchers use PU learning on a dataset from a mobile service provider. While their study shows excellent results they, unlike our work herein, utilize features characterizing the individual handsets and users on the network. These features may make generalizing to classifying network traffic on new networks difficult.

In [39], the authors provide an excellent description of how to implement PU learning. The techniques from [39] inspired some of the approaches and result evaluation used in this chapter.

Combining multiple applications into a single class is a technique used in several traffic classification efforts, though they are of the more traditional forms of traffic (e.g., browsing, email, chat (non-encrypted), or file transfer). For example, research projects described in [40–44] all use a popular dataset (called ISCXVPN2016) that was created by a team from the Canadian Institute for Cybersecurity [45]. Another popular dataset is CIC-Darknet2020 [46], which extends the ISCXVPN2016 dataset with Tor[29] traffic (hence 'Darknet') for each application in ISCXVPN2016 (i.e., there are no new applications added).

The authors of [47] used the CIC-Darknet2020 dataset to conduct a study somewhat closely related to ours as it also examines the ability to detect zero-day applications not used in training. Significantly, their method requires utilizing convolutional neural networks (CNN) as a first step

[29] https://www.torproject.org/

to identify flows falling below a given threshold. These are then assumed to be zero-day applications and clustered with a K-Means classifier. Notably, they then require manual labeling of these clusters by domain experts, which may be difficult to replicate in the real world.

In [48] the researchers examine classifying "unseen" (what we call zero-day) applications using a dataset based on the ISCXVPN2016 dataset. The authors refer to the phenomenon of an "unseen" application as a feature distribution drift, which describes the phenomenon we study in Section 4.4.4. Our results with the Signal application contradicts their suggestion that a model trained on a given application a year ago might not be effective at classifying samples from the current version of the application. As we note in the discussion of our results, however, more work must be done to examine the topic further, as each application and genre will likely vary significantly in regards to how long they can continue to use older data. The authors of [48] note that many published models for traffic classification have poor classification accuracy on applications that have experienced significant feature distribution drive. They then propose a clustering technique for any application determined to be previously unseen.

One of the earliest works in zero-day application classification is [49], where the research team proposes a framework called Robust Traffic Classification. The essence of this framework is the same as many works that have followed. First, they determine if a traffic flow likely belongs to a previously unseen class and then next attempt to classify it. In this work, they implement a so-called "bag-of-flows" technique that relies on the temporal correlation of these unseen applications to known flows.

A subset of the authors of [49] published a slightly earlier work in [50] that was focused specifically on zero-day applications. In this earlier work, the team used three network traffic capture datasets not typically seen in this research, providing a different perspective on network traffic classification. The technique they proposed for classification, with good results on classifying general traffic (not encrypted messaging applications), is very similar to the two-step algorithm we use for PU learning in this work. Namely, they seek to cluster flows and determine which clusters might be potential outliers from known classes. Their subsequent work in [49] uses a similar approach. We found that, while moderately effective in our study, two-step approaches lagged in performance compared to the bagging algorithm we implement. Furthermore, the two-step approach requires extra time and resources to do the initial clustering or classification

whereas our bagging technique does not.

A more recent work, [44], conducts semi-supervised learning to take advantage of unlabeled data for network traffic classification, similar to the work we present. In their work, the authors use deep learning, which is a logical next step to the work presented in this chapter. Like much of the work in this space, the authors of [44] use the bi-flow as their traffic object for classification, feeding their deep learning models packet-level features such as size and direction. These features are more fundamental than those created for and utilized in our work, but deep learning excels at feature extraction and is a notable difference between our works.

4.6 Conclusion and Discussion

Ultimately, real-world use cases demand that ML models trained to classify specific applications would need to detect these applications in traffic from a live network. In this real-world network data, it is almost certain that the model will see "zero-day" applications, i.e., those never seen during training, from the same genre as the target applications. Zero-day applications are highly likely because there are far too many applications to create complete, up-to-date training data for them all. In this chapter, we demonstrate a practical approach to classifying zero-day encrypted messaging applications.

By combining several of the most popular examples of encrypted messaging applications into a single class, we successfully detected hidden zero-day application traffic flows amongst a large set of unlabeled data. We utilized positive and unlabeled learning to take full advantage of the latent information in the unlabeled data. Detecting zero-day applications in this manner is significant because having access to large amounts of unlabeled data and only a small portion of labeled samples is common in real-world classification tasks.

Furthermore, we established that, at least in one encrypted messaging application's case, a difference of nine months does not cause such a significant feature distribution drift that models trained with data from previous versions are useless. This topic requires more exploration of feature distribution drift, and no broad conclusion can be made since each update to an application has the potential to cause profound change. Still, our results in this chapter suggest that organizations need not recreate labeled data when a new version of an encrypted messaging application is released.

In this study, we found that without downstream inter-arrival timing data, our models were less effective at identifying zero-day encrypted messaging applications after having trained on a combined class representing the genre. Using any timing data at all raises a concern that a bias may exist in the timing characteristics of our capture system compared to that of the MIRAGE-19 capture system used to generate the unlabeled data set. To address this issue, we discarded all timing features in the outbound (or upward) direction to eliminate some of the potential bias from our local network medium. At a minimum, future work on this problem should include many other samples generated from non-encrypted messaging applications from the same capture environment. However, the main takeaway from this issue is that it is complicated to perform classification in the real world. Data used to train a model for zero-day detection in the real world will likely have come from a different capture system than the network on which the predictions are made, thus making it hard to control for bias in timing features from the system that generates the training data.

The features we use in this chapter are statistically descriptive, and deep learning models can further improve upon the classification task with a more raw form of the data. Using manually generated features means our models only get access to what we give them. There may be information hidden that we do not expose the model to and thus do not obtain the best results. A more exhaustive feature engineering effort may yield better results, but the time investment is not worth it if deep learning can access that information without manual intervention.

We explored six unique encrypted messaging applications in this study's experiments, and noted that applications tended to trend with a partner. That is, Signal and Telegram performed similarly, at the low end, BiP and Viber performed similarly, at the high end, and WickrMe and WhatsApp were often right around the average. It would be interesting to determine if various subsets of our encrypted messaging applications would yield better zero-day detection on different applications. The combination of the encrypted messaging applications may mute the sensitivity of our model's generalization. These results suggest there are possibly *sub-genres* of encrypted messaging applications, i.e., small groups of applications that all behave similarly. It is possible that a few models trained on these sub-genres would achieve even higher results while still not requiring a new model for every application in the genre.

We find the lower relative performance of Signal and Telegram and, that they trend together, interesting. While difficult to empirically demonstrate, as there are many factors involved, it is our opinion that both of these applications are known to prioritize user privacy and security more so than the other applications in this study. They collect as little user data as possible and do not share this information with third-party advertisers or other partners. Additionally, they are both at least partially open-source code bases. If we were to pick a third application of our six that shares some of these characteristics, it would be WickrMe, which interestingly is the next in order of ascending performance. It's difficult to draw any concrete conclusions from this, but it suggests that perhaps more work is needed to examine precisely what it is that makes these apps perform as they do.

Overall, the work in this chapter establishes that a PU learning model trained on network traffic flows from encrypted messaging applications can generalize with reasonable effectiveness to zero-day applications of that genre. While improving the accuracy is possible with labeled data belonging to that same class, it is unrealistic to expect entities training these models to have up-to-date, complete labeled data on every application in a genre. The techniques presented in this work provide insight into how researchers can accommodate this real-world constraint and achieve acceptable results.

CHAPTER 5
DEEP TRANSFER LEARNING FOR TRAFFIC IMAGE CLASSIFICATION

5.1 Introduction

Network traffic classification plays a critical role in computer networks for various industries. Previous chapters introduced the motivations for studying novel network traffic classification techniques. As a recap, motivations include detecting and preventing security threats, optimizing the performance and reliability of network services, monitoring and troubleshooting network issues, and measuring customers' network usage for accounting and billing. In other words, computer network traffic classification is critical to ensure computer networks' security, performance, and reliability. It also has many applications in various industries, such as telecommunications, finance, healthcare, and government.

A generation ago, classification techniques using IP addresses and port numbers were reasonably effective since network traffic was unencrypted. This gave way to traditional machine learning-based methods and other statistical analyses. Deep learning has become increasingly common in recent studies of network traffic classification, due to advancements in deep learning capabilities and the increasing accessibility of tools for researchers. For example, as noted in [8], these types of models are now commonly used to analyze raw network data. Since deep learning can innately discover complex relationships between attributes of network traffic data, little or no feature engineering is required. This type of feature engineering can be seen in nearly all of the previous network traffic classification research, e.g., [51–54]. While these techniques were often effective, the requirement to heavily preprocess the collected data is a costly addition to an already laborious process of capturing and accurately labeling network traffic data.

Over the past decade or more, deep learning research has focused on topics such as computer vision, natural language processing, and recommendation systems. The relative maturity of deep learning research in these fields helps propel new research as researchers today can leverage many pre-trained computer vision models to accelerate their research progress.

Currently, no such ubiquity of pre-trained deep learning models for computer network traffic classification exists. In fact, it was not until the middle of the last decade that we saw a large

increase in published papers applying deep learning to network traffic classification. Typically the authors of these papers trained neural networks from scratch with an often-used public dataset to conduct classification. While some published work focuses on the transfer of learning for network traffic classification, this work is mostly just between two models built from scratch by a given research team.

In this chapter, we explore the combination of the aforementioned state-of-the-art image classifiers with the relative nascent field of deep learning for network traffic classification. We take highly successful pre-trained models from the computer vision domain, i.e., ResNet, and transfer its inference capabilities to the computer network traffic classification domain. We adapt data from computer network traffic flows to the image domain to bridge these two domains. Specifically, we convert raw packet captures into grayscale images that are then used to fine-tune pre-trained ResNet models. Thus, we leverage the powerful predictive capability of ResNet towards our "images" of network traffic flows. This technique allowed us to take advantage of the mountains of deep learning research in the computer vision domain and transfer it to network traffic classification with surprisingly good results.

We note that most network traffic classification research uses a small set of publicly available datasets. Namely, ISCX-VPN dataset [45], USTC-TFC2016 [43], or CIC-Darknet2020 [46] routinely show up in the literature. While useful in comparing results across research projects, success on these well-curated datasets doesn't always convey that the published methodologies will be successful in live networks where hundreds of applications and network services exist. That is, these datasets contain significantly fewer examples of network traffic than are encountered on a live network. It's been noted in [48] that updates to network applications can also cause a phenomenon dubbed 'feature-distribution drift.' This means that, over time, an application can look so different from a feature perspective that it is essentially a new application. It is, therefore, important to reexamine these traffic classification techniques with modern network data.

To gauge the robustness of our deep learning approach to network traffic classification (and, by extension, perhaps many others), our study uses a modern, live capture from our university's Wi-Fi network to conduct classification research. A significant challenge in using live network traffic data is obtaining ground truth labels for each network flow. We utilize a commercial device (in this case, Palo Alto firewalls at the perimeter of the campus network conducting application

identification) that labels each network traffic flow on a network as belonging to a specific application. The process is not always successful, (i.e., some applications are completely encrypted end-to-end via VPN, or the device isn't sure) but we are using state-of-the-art tools for production network traffic classification. We note that the intelligence used in making these categorizations is not always publicly available, i.e., it's often a trade secret, but generally these tools attempt to identify applications based on obvious parameters such as IP address and port numbers, or use "in the clear" information visible during a TLS session handshake, or apply more complex methods like heuristics. While these tools are state-of-the-art, any labels obtained in this manner must be treated as *quasi-ground truth*; we note, however, in our efforts to validate the labels, they appear reliable.

Finally, as we noted in earlier paragraphs, traditional machine learning can provide excellent network traffic classification results with statistical features about network traffic flows. Several studies, e.g., [24, 55, 56], have shown excellent results for network traffic classification with tree-based classifiers such as decision trees, random forests, or gradient-boosted trees (specifically, the XGBoost [25] implementation). Because deep learning models often require more time and resources to train for network traffic classification tasks, it is worthwhile to consider whether or not increased classification accuracy warrants the extra costs required. Thus, as a point of comparison to our transfer learning approach in this study, we create statistical features describing each of the network flows and perform a standard multi-class classification utilizing the XGBoost classifier on the same data.

The primary contributions of this work include the following. We:

- Describe an approach to represent the payload and inter-arrival time values for individual packets comprising a network traffic flow as 8-bit grayscale images to be used in network traffic classification.
- Demonstrate that transfer learning from a computer vision model (i.e., ResNet) to the domain of network traffic classification is possible and highly effective.
- Conduct a multiclass classification on a large (nearly 10 times as large as other popular datasets), real-world dataset captured from a campus network using these deep transfer learning techniques.

- Provide a direct comparison between the application of deep transfer learning and a traditional machine learning tree-based classifier for network traffic classification on a dataset captured from a live network.

5.2 Background

5.2.1 Creating Network Traffic Images from Bi-Directional Traffic Flows

To take advantage of the ResNet computer vision models, we must first create images of our network traffic flows. Thus, we construct grayscale Portable Network Graphics (PNG) images from our captured network flows in this study. Grayscale PNG images are typically implemented with 8-bit values per pixel, each describing a given pixel's luminance (or brightness). This translates to each pixel being assigned a numerical value within the range of 0-255, where 0 represents a black pixel, 255 represents a white pixel, and everything between is a shade of gray. The following paragraphs detail this traffic image creation process.

Two noteworthy decisions were made and implemented during our initial capture and processing of network traffic flows. First, we keep only the first 32 packets in a network flow. We note that several studies use even fewer than 32 packets, e.g., [57], alleging that most of the 'interesting' information in a network traffic flow occurs early in the conversation. We chose 32 packets as this value exceeds what some studies suggest is necessary, but remains a manageable amount of data that nicely supports grayscale image generation (as discussed in the remainder of this section). Second, any flow which does not have 32 packets is zero-padded to ensure all network traffic flows have the same length, which increases computational efficiency in deep learning. This zero-padding technique is common in nearly all of the literature we have encountered.

In its rawest state, after capturing and processing the campus Wi-Fi data as described in Section 2.2.2, our data is stored in comma-separated value (CSV) files. One file exists per application (or class) in the network capture. In each of these CSV files, a single row containing 64 values represents a single network traffic flow. The 64 values per row represent each of the 32 packets' with (1) application payload size and packet direction and (2) inter-arrival timing value. The payload size for each packet is a signed integer, where a positive number indicates that the payload was part of an upstream message in the bi-directional traffic flow, and a negative number

indicates that the payload was sent in the downstream direction during the flow. We note that the terms upstream and downstream use the reference point of the device within the campus network, such that upstream refers to messages leaving the campus Wi-Fi. Additionally, we note that this paradigm of upstream and downstream with the campus network as the point of reference works only because we discard any intra-campus network traffic flows. In other words, no traffic flows have both the source and destination IP address within the campus network boundary. Lastly, any packet with a payload length of zero indicates the packet is a control message for the Transmission Control Protocol (TCP) (e.g., acknowledgment or reset packets) and does not describe the *behavior of the application in question*. Thus, TCP control packets are discarded.

In this work, we want a balanced dataset with 10,000 samples per class from our live network capture, which requires us to upsample or downsample various classes as some classes have less than this target and many have far more. When a class has too few samples, the upsampling process can result in our model overfitting those classes, ultimately hurting our overall accuracy metrics for the other classes. Thus, to avoid drastically upsampling the same small amount of samples to our target of 10,000, any class with fewer than 1,000 unique network traffic flows is discarded before converting the data to images. After discarding these classes we are left with 172 unique applications, listed in Table 5.1.

The next step in creating our network traffic images is to rescale the payload and inter-arrival time values to be properly displayed in 8-bit grayscale, i.e., as an integer between 0 and 255. We apply a min-max scaling to each packet's payload value. Across the entire dataset, the largest application payloads are 1472 bytes. As mentioned in the previous paragraph, we negate the payload value if the device received the packet in a downstream direction. Therefore, the range of possible values for payloads across all packets in the dataset is given by the set $\{x \in Z \mid -1472 \leq x \leq 1472, \text{ where } x \neq 0\}$. These 2944 discrete values for payload length are then scaled into 256 luminance values, which results in a small amount of "information loss," i.e., there is one luminance value for every 11.5 payload values. We believe that this information loss actually improves the models trained on this data; that is, instead of learning that a series of packet sizes were, for example, $[2, 1, 4]$, the model would instead learn that there were three very small packets back-to-back (i.e., all three packets would receive the same pixel luminance value).

Table 5.1 Shown in this table are the 172 applications categorized by the Palo Alto firewalls used in this study, all of which had more than 1,000 unique network traffic flows captured.

adobe-creative-cloud	geforce-now	mqtt	steam
amazon-echo-conntest	giphy	ms-delve	stun
amazon-fire-tv-conntest	github	ms-office365	synology-cloudstation
amazon-music	gmail	ms-onedrive	taobao
amazon-prime-video	google-analytics	ms-product-activation	teamviewer
appdynamics	google-base	ms-store	telegram
apple-game-center	google-chat	ms-update	tesla-car-app
apple-maps	google-docs	naver-line	tiktok
apple-push-notifications	google-drive-web	netflix	tinder
apple-siri	google-maps	new-relic	tokbox
apple-update	google-meet	nexon-launcher	traceroute
apt-get	google-messages	nintendo-conntest	trello
arcgis	google-play	nordvpn	trendmicro
avast-av-update	google-update	notion	tumblr
avira-antivir-update	gotomypc	ntp	twitch
aws-iot	grammarly	ocsp	twitter
bing-maps	hbo	open-vpn	ubisoft-uplay
bitdefender	hotmail	origin	unknown-tcp
bittorrent	http-proxy	overwatch	unknown-udp
boxnet	http-video	pandora	untunneled
brightcloud	hulu	pinterest	viber
canon-bjnp	ichat-av	playstation-network	vimeo
canvas	icloud	qq	vudu
cisco-spark	imap	quora	wargaming.net
clash-of-clans	imgur	reddit	web-browsing
cloudinary	incomplete-flow	ring	webex
coinbase	instagram	roblox	webroot-secureanywhere
dailymotion	itunes	rocket-league	websocket
datadog	jamf	rtp	wechat
dcinside	jumpshare	rustdesk-remote-desktop	whatsapp
deezer	kakaotalk	samsung-updates	wickrme
discord	kaspersky-ksn	signal	windows-azure
disneyplus	kaspersky-netagent	sina-weibo	windows-defender-atp
disqus	lastpass	slack	windows-push-notifications
dns	league-of-legends	smtp	wireguard
dns-over-https	linkedin	snapchat	xbox-live
dns-over-tls	liveperson	snmp	yahoo-calendar
dropbox	mail.ru	soap	yahoo-mail
dynatrace-app-monitoring	mailchimp	soundcloud	yahoo-web-analytics
evernote	malwarebytes	source-engine	yelp
facebook	mega	speedtest	youtube
facetime	mendeley	spotify	zendesk
foursquare	minecraft	stackoverflow	zoom

Like payload sizes, inter-arrival times are also min-max scaled to the 0-255 range. Due to the extremely large range of inter-arrival values, we first do an intermediate logarithmic scaling before we apply the min-max normalization. Since inter-arrival time values are stored in microseconds and some network traffic flows can (theoretically) last for days, the range of inter-arrival values is quite-large. In addition, the range can have extremely large outliers. Thus, if we were to apply min-max scaling to the inter-arrival time values without this logarithmic scaling step, then nearly every pixel that is not an outlier becomes 0 (or total black). In other words, we achieve much better utilization of the grayscale by applying a logarithm to each inter-arrival time value before min-max scaling. The relative spread of pixel values is then more uniform because the exponentially larger timing values are scaled down.

Finally, after both payload and inter-arrival time values are scaled to 0-255, we create the image by writing those values to individual pixels. Given that we have 64 total values per flow, we arrange the pixels in an 8x8 square, with the top half (the first 32 pixels) representing our scaled payload values and the bottom half (the last 32 pixels) representing the scaled inter-arrival time values. The Python Imaging Library (PIL)[30] is used to write the values to disk in the specified PNG format.

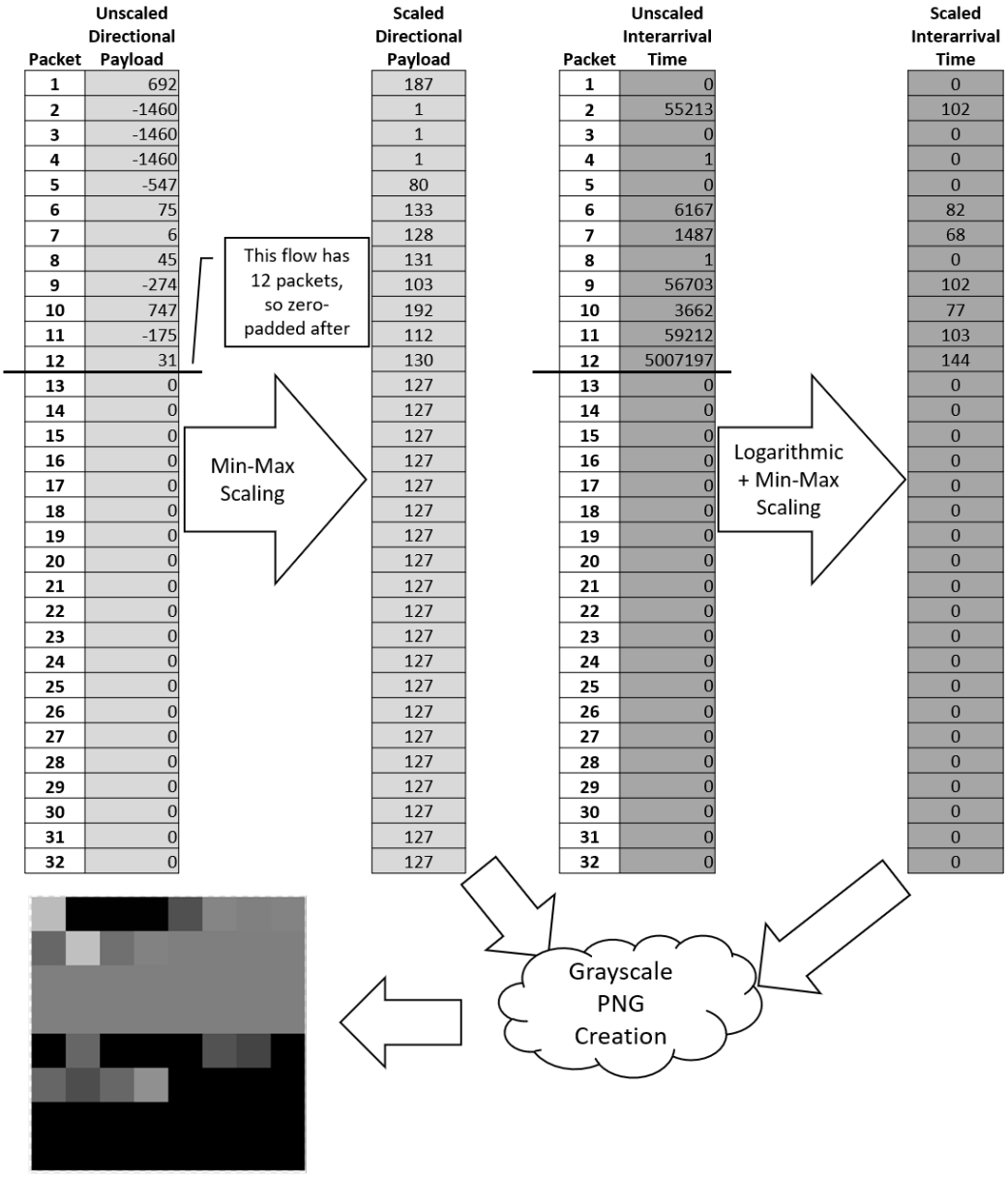
We note that all scaling operations are identically applied across the entire dataset instead of per-flow or per-application, i.e., the minimum and maximum values used to min-max scale individual payload and inter-arrival time values are the same for *every flow* of *every application*. Thus, many flows will not have completely white or black pixels as their specific payload and timing values fall between the global minimum and maximum values. We illustrate our end-to-end grayscale image creation process in Figure 5.1. Visual examples of a few of our network traffic images are shown in Figure 5.2.

5.2.2 ResNet Model Selection

In this study, we aim to transfer the learning from a neural network trained for image recognition to network traffic classification, a type of domain adaptation. Convolutional Neural Networks (CNNs) are generally recognized as the best deep neural networks for computer vision tasks, due to their ability to extract meaningful features from image data. A specific type of

[30]https://pillow.readthedocs.io/en/stable/

CNN, ResNet, has achieved state-of-the-art performance on several computer vision tasks, such as image classification, object detection, and segmentation. As such, we utilize ResNet as our base model for transfer learning in this study.



Packet	Unscaled Directional Payload	Scaled Directional Payload	Unscaled Interarrival Time	Scaled Interarrival Time
1	692	187	0	0
2	-1460	1	55213	102
3	-1460	1	0	0
4	-1460	1	1	0
5	-547	80	0	0
6	75	133	6167	82
7	6	128	1487	68
8	45	131	1	0
9	-274	103	56703	102
10	747	192	3662	77
11	-175	112	59212	103
12	31	130	5007197	144
13	0	127	0	0
14	0	127	0	0
15	0	127	0	0
16	0	127	0	0
17	0	127	0	0
18	0	127	0	0
19	0	127	0	0
20	0	127	0	0
21	0	127	0	0
22	0	127	0	0
23	0	127	0	0
24	0	127	0	0
25	0	127	0	0
26	0	127	0	0
27	0	127	0	0
28	0	127	0	0
29	0	127	0	0
30	0	127	0	0
31	0	127	0	0
32	0	127	0	0

Figure 5.1 Our end-to-end traffic image creation process for a single network traffic flow from the Minecraft application is depicted in this figure. As shown, the inter-arrival times have a logarithm applied before min-max scaling to the 0-255 range. After min-max scaling is completed, each of the 64 scaled values is translated to an individual pixel in the final grayscale PNG image of the network traffic flow. We note, the flow selected in this figure only had 12 non-zero-length messages exchanged. Thus, packets 13-32 are zero-padded to ensure a length of 32 for grayscale image creation.

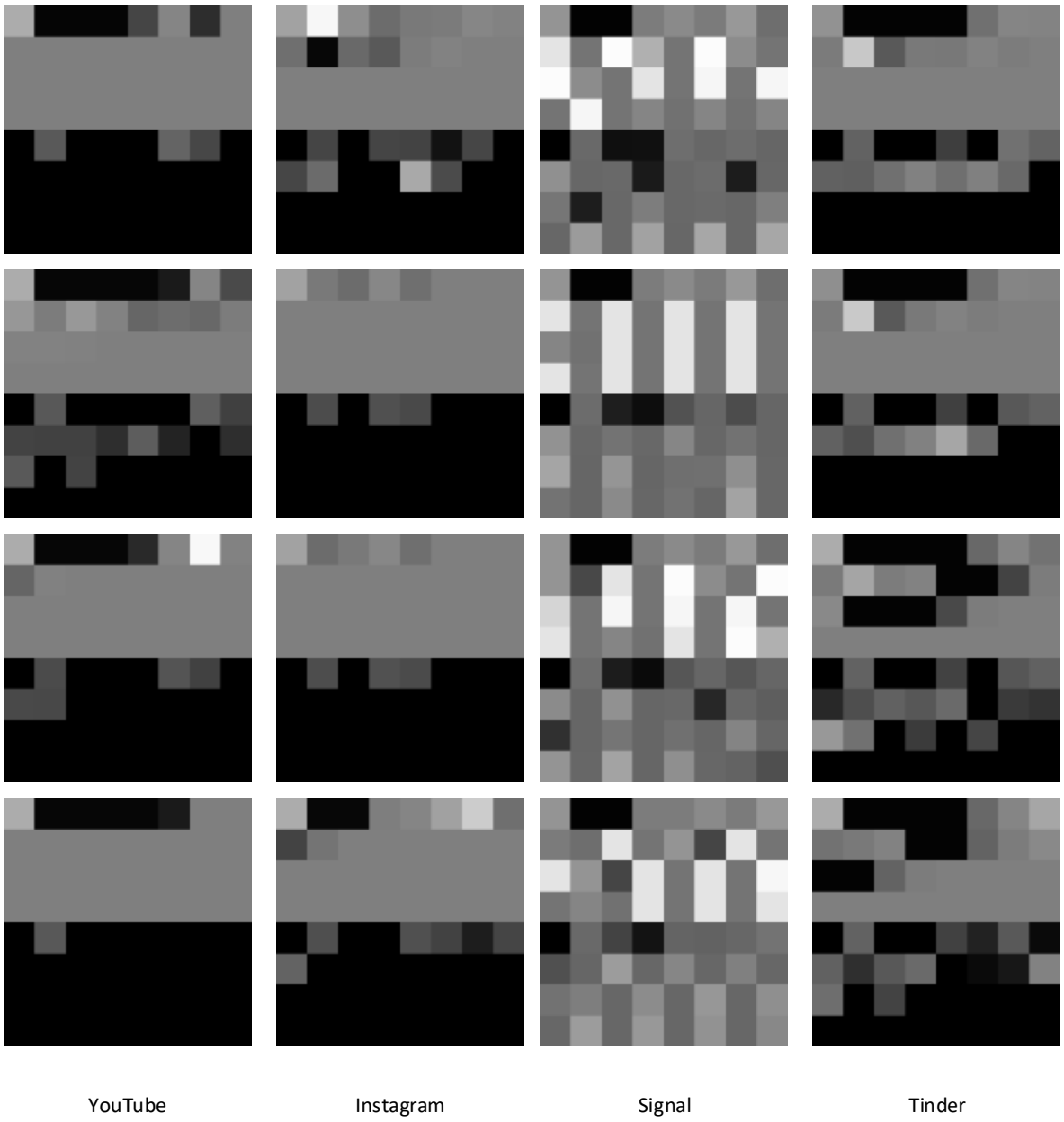


Figure 5.2 This figure visualizes four applications in our dataset after traffic image creation: YouTube, Instagram, Signal, and Tinder. Each application has four sample images in the column directly above its label. These examples were hand-selected to highlight how, even with the human eye, clear patterns in the images represent network traffic flows. Of course, not all flows are as obvious as the ones shown but, generally, a few distinct patterns can be seen in the traffic images for each application. This visual inspection of the data suggests that, if humans can see patterns, deep learning models such as ResNet will as well.

The ResNet (short for "Residual Network") architecture was first proposed in [58]. This paper introduced the concept of residual learning, which involves adding residual connections that skip one or more layers in a neural network, to facilitate the training of very deep networks. The ResNet architecture has since become one of the most widely used and successful deep learning architectures, particularly in computer vision tasks such as image classification and object detection. This popularity and success make it an ideal candidate to transfer to the task of network traffic image classification.

Utilizing the fastai deep learning library [59], several pre-trained ResNet models first described in [58] are available. ResNet18 has 18 layers, consisting of 16 convolutional layers (organized into several blocks, each containing multiple convolutional layers with different filter sizes and numbers of output channels), one fully connected layer (used at the end of the network to combine the extracted features and generate the final output), and one pooling layer (used to reduce the spatial resolution of the feature maps and capture the most important features in the input image). It is designed to be a relatively small and efficient network while still offering good performance on image recognition tasks. The other variants in [58] and available in [59] include ResNet34, ResNet50, ResNet101, and ResNet152, which have 34, 50, 101, and 152 layers, respectively.

With all other parameters (batch size, learning rate, etc.) being held constant, adding more layers to a ResNet architecture increases training time. The benefits of deeper networks in increased accuracy and improved feature representation, as you would expect, can outweigh the extra training time cost. Our initial experiments focused on determining which ResNet variants provided reasonable accuracy without prohibitive training times. With our dataset, the ResNet50 model offered a good balance between training time and classification performance. Thus, the ResNet50 model is our choice for this study. The ResNet50 model is the "middle-of-the-road" concerning the five ResNet variants, leaving us unsurprised that we found it the most balanced in training time versus accuracy.

The ResNet models provided in [59], like most pre-trained ResNet models, are trained on the ImageNet dataset [60], a large dataset consisting of 1 million images with labels in 1000 different categories. When pre-trained ResNet models are used for either domain adaptation or fine-tuning (two different but related types of transfer learning), it is common practice to initialize the

network weights with the parameters learned from the ImageNet dataset. This common practice helps speed up convergence and improve performance. In this study, as described in the previous section, our images are representations of network traffic flows. This fact makes the transfer learning task significantly more challenging, as we seek to establish how well the pre-trained ResNet models can learn the "shapes" of our network traffic flows.

5.2.3 Statistical Features for Network Traffic Flows

We describe the generation of statistical features from our per-packet representation of network traffic flows in this section. As discussed in Section 5.2.1, each network traffic flow is stored as 64 comma-separated values. The first 32 are signed integers representing the direction and payload size, and the last 32 are the inter-arrival timings between packets in the flow. We create tabular data from these values by iterating over each traffic flow and calculating various statistically descriptive features.

The features we generate per traffic flow are the same as those in Section 3.3, with a small exception. In this experiment, our raw packet data contains the application payload size instead of the total IP packet lengths. In other words, we do not use the IP header length, which typically does not vary greatly and, thus, adds little information for models to learn. Each statistical feature is calculated for payloads and inter-arrival timings in the upstream, downstream, and bi-directional interleaving. Calculating statistics for each of these directions provides our model with as much information as possible (and is common in much of the literature, e.g., [6, 9, 61]). Conversely, in the deep learning approaches, we rely on neural networks to extract these features without us manually engineering them. The statistical features used for this experiment are listed in Table 5.2.

An interesting phenomenon occurs when generating statistical features in this way. Because statistics are calculated in different directions, certain flows receive no values for some metrics. For instance, if a network traffic flow was a series of three packets in the outbound, or upstream, direction, then there is no statistical value for the downstream direction. Further, if a network traffic flow has just one upstream packet, then it has no inter-arrival time values. We do not engineer any features with the traffic image representation in Section 5.2.1 as we see each payload and timing value as an individual pixel. Thus, representing flows as traffic images yields no

missing values. We discuss the implications of this further in Section 5.3.2

Table 5.2 Features calculated for each traffic flow are shown in this table. Each feature is calculated for (1) two attributes (payload size and inter-arrival timing) except for *num_packets*, *app_payload_bytes*, and *duration*, and (2) three directions (upstream, downstream, and bi-directional).

Feature Name	Directions × Attributes	Number of Features	Feature Description
num_packets	3×1	3	Number of packets
app_payload_bytes	3×1	3	Total bytes of application payloads
duration	3×1	3	Flow duration in seconds
min	3×2	6	Minimum
max	3×2	6	Maximum
mean	3×2	6	Average
std	3×2	6	Standard Deviation
var	3×2	6	Variance
mad	3×2	6	Mean Absolute Deviation
skew	3×2	6	Skewness
kurtosis	3×2	6	Fisher Kurtosis
x_percentile	3×18	54	Value at each of 9 deciles (10 through 90)
BF_label	N/A	1	Application label for bi-flow
Total		112	111 features + 1 label

5.3 Overview of Our Work

5.3.1 Transferring ResNet to Traffic Image Classification

In this section, we provide an overview of the experiment conducted in this study (see Figure 5.3). Because our dataset is captured from a live network, the classes represented are naturally imbalanced. While deep learning with imbalanced data is possible, having a balanced dataset (where each class has the same, or close to the same, number of samples) ensures models can learn to recognize different classes or categories in the dataset and avoid bias towards any particular class. As first described in Section 5.2.1, we upsample or downsample each of the 172 classes to 10,000 network traffic images to achieve our balanced dataset. In some cases, if the class has less than 10,000 unique samples, this requires upsampling (i.e., some samples will be repeated in the 10,000) otherwise we downsample to 10,000. With 172 classes, this results in 1.72 million ($172 \times 10,000 = 1,720,000$) network traffic images used in this study. As a point of

reference, the widely used ISCXVPN2016 dataset [45] has roughly 1/10th as many traffic flows in roughly 20 classes belonging to 15 different genres.

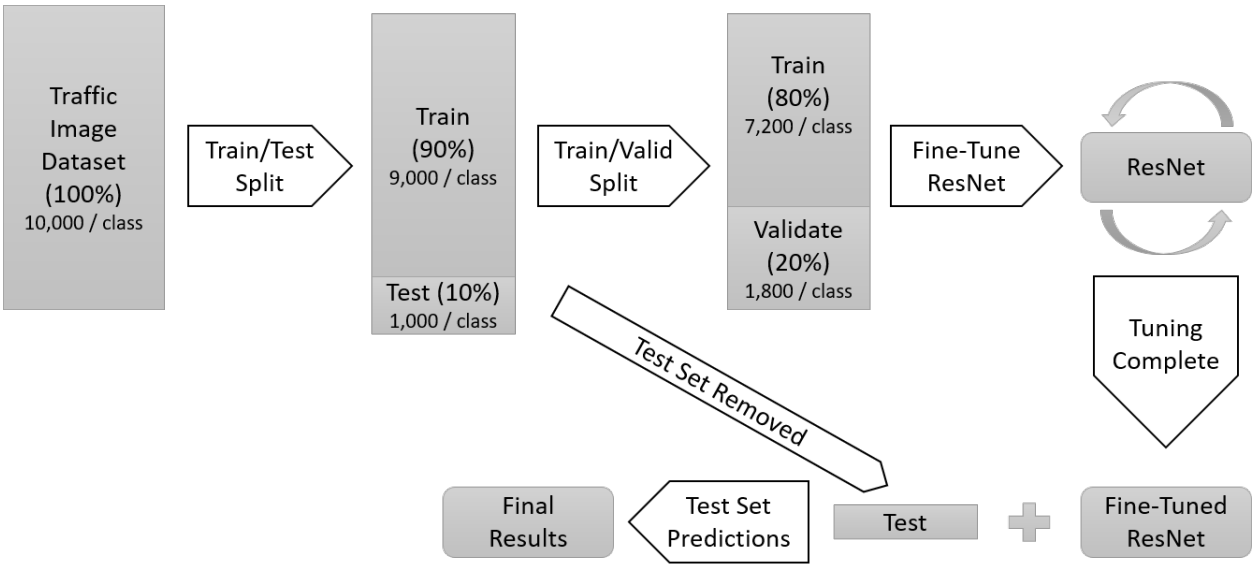


Figure 5.3 This figure illustrates the overall data splitting and training process for our deep transfer learning algorithm. The Traffic Image Dataset, which has 1.72 million images, is subject to a 90%/10% train/test split. The 90% of the data reserved for training is further split into 20% validation and 80% actual training data. The actual training data is used to fine-tune the ResNet50 model until the validation set's cross-entropy loss converges. The resultant model is used to predict against the test set that was withheld from training.

We split our data into training, validation, and testing sets to evaluate our model's performance properly. Testing data accounts for 10% of the original data, while training and validation accounts for the other 90%, which is further divided into an 80%/20% split between training data and validation data. In other words, 10% of the data is used for testing, 18% for validation, and 72% for training. The key difference between training and validation data is that training data is shown to the model during each training epoch, while validation data is only used to evaluate the model's performance and tune hyperparameters (such as the learning rate) during training. By using separate datasets for training and validation, we avoid overfitting the training data and ensure that the model can generalize well to new, unseen data. The testing set is, in some ways, similar to the validation dataset in that it is never shown to the model for training (as it is used for evaluating the model's performance). The main difference is that we do not make predictions against the testing dataset until *after training is complete.* That is, no feedback

system utilizes the test set's results during training, as with the validation data.

For this study, we implement our deep learning models with fastai [59], a deep learning framework built on PyTorch. The framework provides a high-level API that quickly and easily trains deep learning models. As highlighted in Section 5.2.2, we download a pre-trained ResNet50 model as the base model for our transfer learning experiments in this study. Our overarching goal in this chapter is to fine-tune the ResNet50 model for our network traffic image classification task.

ResNet, a CNN, comprises many linear layers with nonlinear activation functions culminating in a fully connected layer with softmax output. This final output provides a prediction probability of the sample belonging to each class the model used in training. For example, if there are 10 possible classes, the output is an array of 10 values, all summing to one, with the highest value being the model's predicted class. In the case of the ResNet50 models, this final layer has 1,000 outputs because there were 1,000 classes used in ImageNet during training. Since our dataset does not contain these 1,000 ImageNet classes, we replace the final layer with a new one containing 172 outputs, one for each traffic image class (the 172 classes listed in Table 5.1). The weights for this final layer are randomly initialized.

One often recommended image size for ResNet50 is 224x224 pixels, which is significantly larger than our 8x8 traffic image format. The fastai library provides adaptive pooling capabilities that allow models to accommodate smaller image sizes; however, we elected to resize our 8x8 images to 224x224 during the training process. This resizing ensures we obtain the best possible accuracy from the ResNet50 model.

The next step is fine-tuning the model with our newly added final layer. The challenge in transfer learning is training the new model without overwriting the pre-trained weights that are provided with our downloaded ResNet model. To accomplish this, we implement what is known as *model freezing*, where every layer except our newly added layer has its weights frozen (i.e., they are not updated) during a specified number of training epochs. Generally, the more diverse the target task is from the source task on which the model was originally trained (in our case, it's considerably different), the larger the number of frozen epochs is warranted. After conducting an iterative parameter tuning, we settle on 10 epochs for fine-tuning ResNet with only the final layer unfrozen. During these 10 epochs, our model takes advantage of the carefully tuned parameters that are provided with the pre-trained ResNet50 model while still adapting the model to our task.

After training for 10 epochs with the pre-trained layers frozen, we train the entire model with unfrozen weights. In [62], it is shown that initial layers in a CNN capture broad concepts, such as detecting edges and gradients, which is highly relevant for our task of traffic image classification. In contrast, the subsequent layers learn more specific features (e.g., eyes), which are unimportant to our classification task. Because of this fact, we use discriminative learning rates when training the model in an unfrozen state. In other words, earlier layers are trained with a lower learning rate to avoid deviating too much from the pre-trained weights in these generally useful layers. Later layers, by contrast, have a higher learning rate as the specific features of images in our target data are much different than those the ResNet50 model was trained on. Therefore, these later layers will likely need a greater change to the pre-trained weights. Even though we have unfrozen the entire model, discriminative learning rates also help preserve the pre-trained values that are most valuable to our task.

Additionally, during training, we implement a relatively new regularization technique known as MixUp [63]. MixUp is an effective regularization technique, particularly for improving the generalization performance of deep neural networks on classification tasks. MixUp allows the model to learn a more general decision boundary that is less prone to overfitting the training data by mixing two randomly chosen images and their labels. Additionally, MixUp has been shown to improve the model's ability to handle out-of-distribution samples, which differ significantly from the training data. In every case, MixUp improved the validation loss for the experiments in this study. However, we note that it does increase the loss for the training data, which is not uncommon for regularization techniques that prevent overfitting to training data.

Finally, we also implement early stopping in our training to avoid wasting time and resources and potentially overfitting our training or validation datasets. We monitor the cross-entropy loss of our validation dataset as the stopping criterion. Once five epochs have passed with no further improvement in our validation loss, we stop training. The model with the last best validation loss is then chosen. That is, once we determine we have hit a plateau in the validation loss, we recover the last model that saw a validation loss improvement. Throughout our experiments, the models typically train for 40 to 60 epochs before reaching this early stopping criterion. In just a few cases, our models trained for the chosen maximum of 100 epochs without reaching the early stopping criterion (i.e., the validation loss continued to improve from epoch to epoch).

Once training is complete, we save the model and evaluate its performance on the test set. The model is asked to make predictions on the 10% of the original data not used in training. Predictions are evaluated against original labels and metrics are calculated, finalizing our model's performance. Because the validation set is meant to mimic the test set, we expect that the results on the test set should be somewhat similar to that of the validation data.

5.3.2 XGBoost Classification on Statistical Features

In [8], a survey of deep learning for mobile traffic classification, the authors note better global performance when using a Random Forest classifier on statistical features when compared to several deep learning models that use network traffic data in either a one-dimensional or two-dimensional (as we do in this study) format. On average, Random Forest outperformed all the deep learning algorithms and methods though, in some cases, individual deep learning techniques had the highest reported accuracy.

Considering these findings, we evaluated a tree-based classifier with our dataset as a point of comparison to our deep transfer learning method. Rather than utilize the traffic images we created in Section 5.2.1, we generate statistical features for each traffic flow as described in Section 5.2.3. The traffic flows used in this study are the same as those used in Section 5.3.1, in order to create as direct of a comparison as possible. These new statistical representations are split into training, validation, and testing datasets in the same ratios as the network traffic images.

While preprocessing the statistical representations of network traffic flows, as we alluded to in Section 5.2.3, we were confronted with a small percentage of flows containing NaN (not a number) values as a result of being unidirectional. As an example, consider a unidirectional upstream flow that contains three packets; there are no downstream inter-arrival time values for this particular flow. Because XGBoost classifiers (discussed in the following paragraph) require that there are no missing values, researchers can either drop samples that are missing values or impute the missing values. In this study, we elect to drop flows that are missing values as imputing missing values doesn't make sense for our dataset (i.e., if there are no downstream packets in a given flow, it does not make sense to estimate or calculate an average downstream inter-arrival time from other flows in the class).

We utilize the XGBoost [25] classifier in this study as, like Random Forests, it is a tree-based classifier. Also, several studies, such as [9, 24, 64], demonstrate that XGBoost outperforms Random Forests in the vast majority of cases. We use XGBoost, the best available technique, because the goal of this work is to compare how well a traditional machine learning model performs at classifying network traffic flows compared with our deep transfer method detailed in Section 5.3.1. As with the ResNet model fine-tuning, we allow our XGBoost classifier to train for up to 100 epochs. Similarly, we implement an early stopping criterion in case the validation loss shows no improvement over several subsequent epochs.

5.4 Details and Results

In this section, we present more details concerning the configuration of our model in addition to results from each of our experiments. While we provide results for the training, validation, and testing datasets, we are primarily interested in the testing dataset results as they provide the best indicator of how well our model will perform on samples it was not exposed to during training.

5.4.1 Traffic Image Classification Results

In deep learning, the batch size is an important hyperparameter to tune before training as it can impact the training process' accuracy, speed, and stability. Larger batch sizes speed up the training process by reducing the number of batches, but may also result in overfitting; on the other hand, a smaller batch size improves generalization but may slow down the training process. Our study evaluated batch sizes of 32, 64, 128, 256, 512, 1024, and 2048. We found that 64 was most the suitable for our particular dataset, yielding a high accuracy, decent training times, and acceptable cross-entropy losses for the training and validation sets. A batch size of 128 was close in performance to a batch size of 64, and typically faster, but we opted for the improved performance that a batch size of 64 provided.

As described in the previous section, we use lower learning rates for early layers and higher learning rates for subsequent layers in our ResNet50 model. We evaluated weight decay parameters ranging from $1e^{-}2$ to $1e^{-}7$ and found some impact on accuracy results with no clear value as the best; therefore, we set this value to $1e^{-}2$, the default in the fastai deep learning library. The final hyperparameter, training epochs, is given in two values: the number of training

epochs with the pre-trained weights frozen and the number of training epochs with the entire model unfrozen. After analysis, we elect to train for 10 epochs with pre-trained weights frozen and up to 100 epochs with the entire model unfrozen. When the validation loss does not decrease for five epochs, we assess the model as being fully trained.

We report our model's accuracy and f1-score metrics on the training, validation, and testing datasets. Because our dataset is balanced, accuracy is an appropriate metric to evaluate our model's performance as it indicates whether it performs well in correctly classifying both the positive and negative examples. The f1-score is also valuable to calculate, as it indicates how well the model correctly identifies positive samples while minimizing the number of false positive and false negative predictions.

Our model continuously obtains lower validation losses, as desired and expected, in each epoch until training reaches 56 epochs, i.e., we no longer observe a decreasing validation loss for the five epochs following the 56th. The training and validation losses after convergence are 0.849 and 0.223, respectively. Figure 5.4 depicts the training and validation losses during training. We note that it is not common for the validation loss to be lower than the training loss in many deep learning applications. We believe the use of MixUp provided the atypical result shown. That is, the introduction of MixUp, as described in 5.3.1, had two significant effects. First was that our models performed much better on the validation datasets, i.e., they achieved much lower loss results. By combining pairs of examples from the training set, MixUp creates new synthetic examples that are different from the original training examples. This helps to increase the diversity of the training set and improve the robustness of the model to variations in the input data. Moreover, MixUp can also act as a form of regularization, by encouraging the model to learn more smooth decision boundaries that generalize better to unseen data. Second, the training loss values did not drop as rapidly as they did without MixUp. Very low training loss values indicate that a model has overfit to the training data, which is generally unfavorable. By introducing MixUp, we have introduced additional noise and complexity into the training data, which can make it more difficult for the model to fit the data perfectly, resulting in a worse (higher) training loss. However, it's important to note that the goal of training a deep learning model is not to minimize the training loss, but to minimize the generalization error on unseen data (i.e., the validation and testing data).

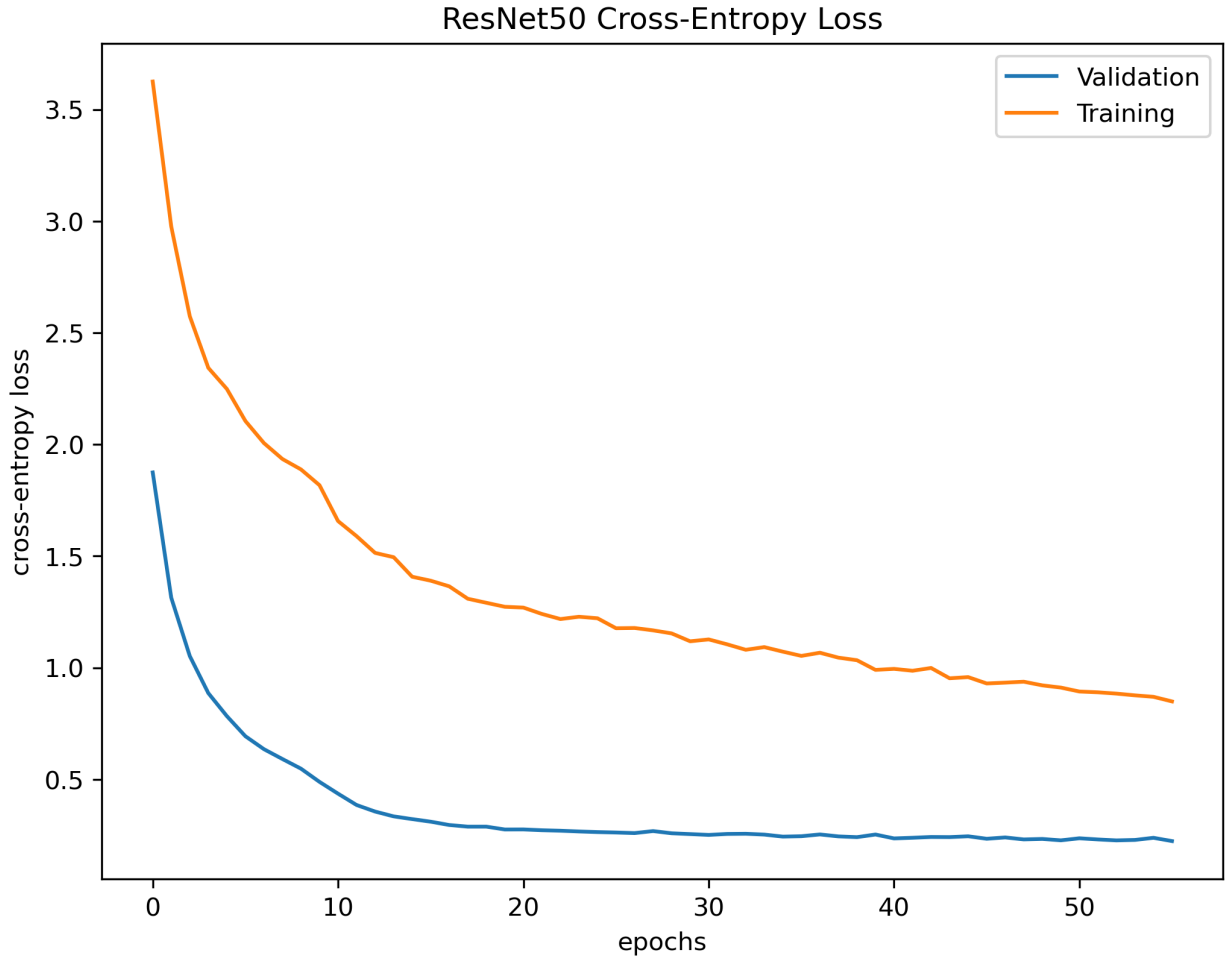


Figure 5.4 This figure illustrates the validation and training loss results, which shows how our model converged during training. After 56 epochs, the validation loss does not decrease for the next five epochs. Thus, early stopping is triggered, and the model at the last best epoch is used. Leveraging the regularization technique of MixUp makes it much more difficult for our ResNet50 model to overfit the training data, resulting in a higher loss for the training data than the validation data.

Table 5.3 This table shows accuracy and f1-scores for the domain adaptation of the ResNet50 model trained on ImageNet to the network traffic classification with our network traffic images.

Dataset	ResNet Accuracy	ResNet f1-score
Training	0.9712	0.9754
Validation	0.9443	0.9427
Testing	0.9547	0.9638

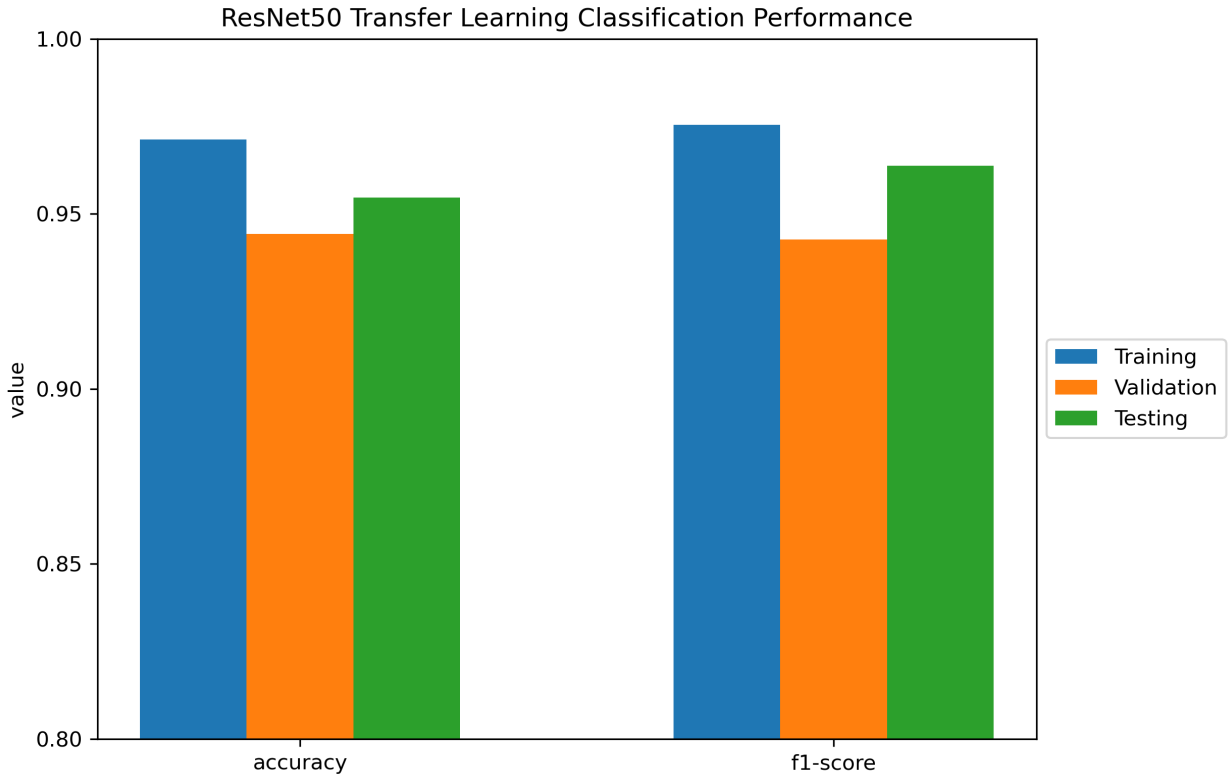


Figure 5.5 This figure shows the accuracy and f1-score results for the ResNet model after fine-tuning for domain adaptation to our network traffic images. Note that the Y-axis is scaled from 0.8 to 1.0 in order to provide more visual resolution when comparing the results. We expect testing and validation results to be similar as the validation dataset is meant to mimic the testing dataset. Training accuracy is provided for completeness. Because the model is given training data during training, we expect it to have higher accuracy than the validation and testing accuracy. See Table 5.3 for details on the results for the ResNet50 model.

Table 5.4 The top five most confused applications in the test dataset by our transferred ResNet50 model. The number of occurrences is out of 1,000 test samples per class. These top five most confused applications represent just over 9% of the total errors.

PREDICTED	ACTUAL	# CONFUSED
amazon-echo-conntest	amazon-fire-tv-conntest	319
google-drive-web	google-chat	117
ms-update	ocsp	112
facetime	ichat-av	93
ichat-av	facetime	87

On our validation dataset, the model achieves an accuracy of 0.9443 and an f1-score of 0.9427. For the test dataset, the model achieves 0.9547 accuracy and 0.9638 f1-Score. Having a higher

test set accuracy than validation set accuracy is not a concern as long as the values are similar, as they are in this case. The validation set is meant to resemble the test set so, if properly constructed, should have similar, though not necessarily better, results. These results demonstrate significant learning took place and our model is able to classify most of the samples in our dataset accurately. As we will discuss in the following paragraphs, some of the errors in our predictions are perfectly reasonable given the similarity of certain classes in our dataset. The training, validation, and test dataset metrics are listed in Table 5.3 and visualized in Figure 5.5.

Typically a confusion matrix would be reported for a classification task like ours, in order to illustrate which classes were the most inaccurately predicted and which class they were confused with. With our 172 classes, such a matrix is difficult to interpret visually. Alternatively, we present the top-5 most confused classes in Table 5.4. In total, the top-5 most confused classes represent approximately 9.3% of the total prediction errors made by our ResNet50 model on the test data. Notably, the most confusing class is connection tests from Amazon Fire TV devices with connection tests from Amazon Echo devices. This confusion seems reasonable, given that both are connection tests from, specifically, devices made by the same company. The second pair of most confused applications belongs to Google. Google Chat was often confused with the web-based Google Drive application. The third most confusing application was the Online Certificate Status Protocol (OCSP), which is used to verify the validity of digital certificates used in online communication. OCSP was most confused with Microsoft Update, which periodically checks for and downloads updates to Windows computers. It's not immediately obvious what these two applications have in common, but perhaps it's related to periodic checks and downloads of updates for certificate revocation lists and other updates. Finally, we note that the 4th and 5th most confusing applications were Apple's FaceTime with iChat-AV and vice-versa. iChat-AV is the predecessor to Apple's iMessage and FaceTime[31], so it is not surprising that these network traffic flows are similar. In Figure 5.6, we show a subsample of the actual traffic images for the classes most confused, illustrating how they share similar patterns to the human eye.

[31]https://en.wikipedia.org/wiki/IChat

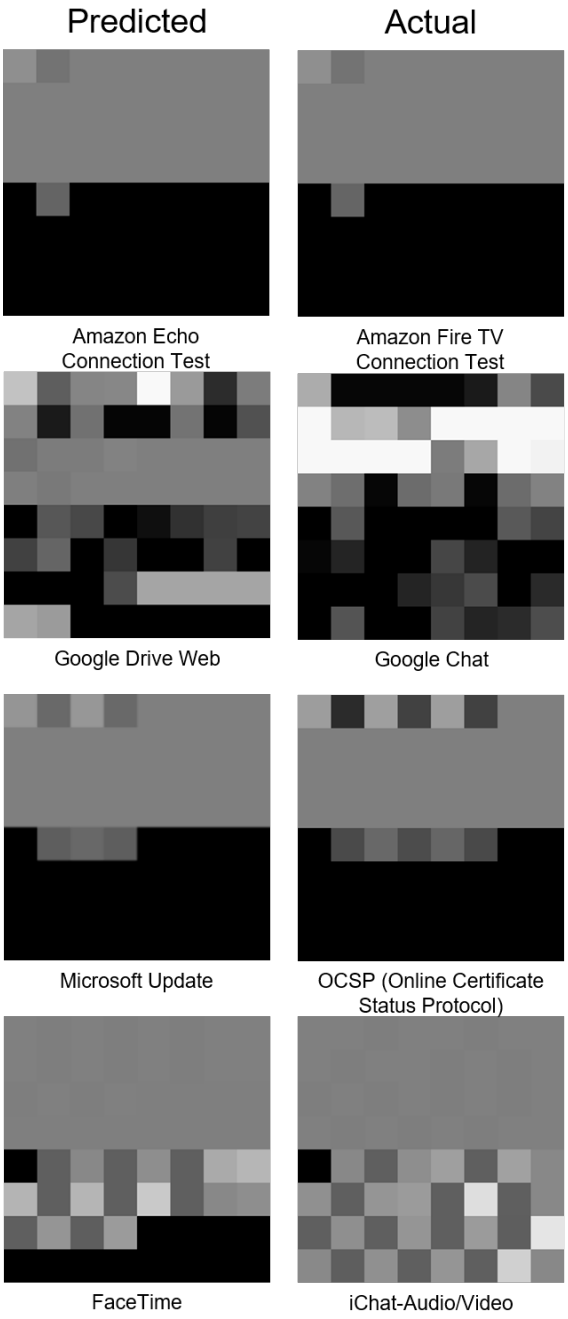


Figure 5.6 This figure visualizes the most confused classes by our fine-tuned ResNet50 model. The 4th and 5th most confusing classes were FaceTime for iChat-AV and vice-versa. Thus, we only show these two classes once in this figure. While these samples were hand chosen to illustrate a human eye can detect patterns in the images, we note that many samples belonging to the depicted classes share a similar pattern.

5.4.2 XGBoost Statistical Feature Classification Results

To attain the best results possible with the XGBoost classifier, we first conduct parameter tuning to ensure optimal settings during training. Tree-based classifiers are prone to overfitting training data, namely if the tree is allowed to grow too deep. As such, our parameter tuning largely aimed to maximize classification accuracy on the validation and, by extension, the testing dataset while ensuring that we did not overfit the training data. We performed a randomized search of several hyperparameters, including maximum tree depth and minimum child weight, which helped prevent a tree from becoming too deep or splitting excessively.

With the hyperparameters selected, we then trained our model for 100 epochs. Our XGBoost model evaluates the cross-entropy loss of the validation set to determine when the model is optimized during training. After 100 epochs, the validation loss had sufficiently stabilized, indicating our model had converged (i.e., the model did not trigger the early stopping criterion prior to 100 epochs, but the epoch-to-epoch gains in validation loss were very minimal in the final epochs). Plots of the training and validation losses for our XGBoost model are shown in Figure 5.7. Because we do not implement the MixUp regularization technique in our XGBoost model (as it is not applicable, though we do utilize L1 and L2 regularization), our training loss is lower than our validation loss.

Similar to the previous section's traffic image classification experiment, we predict against the validation and testing datasets to obtain performance metrics for the XGBoost model. While losses were calculated for the validation set during training, no validation samples were provided to the model to learn from. The testing dataset was also not used by the XGBoost classifier during training.

We report the accuracy and f1-Score for our XGBoost classifier just as we do for the transfer learning model presented in Section 5.4.1. Our XGBoost model achieved a training accuracy of 0.9571 and an f1-score of 0.9569. The validation and testing datasets performed similarly, with the testing dataset receiving slightly more accurate predictions. The purpose of the validation dataset is to provide an estimate of the model's performance on unseen data. Therefore, we expect the validation and testing performance values to be close (assuming we properly avoided overfitting the validation set). It is not uncommon for results on the testing dataset to be slightly higher

than the validation set. In this XGBoost study, the accuracy scores were 0.8852 and 0.8920 for the validation and testing datasets, respectively. The f1-Scores for the validation data was 0.8849 and the testing data is 0.8905. These results are listed in Table 5.5 and depicted in Figure 5.8.

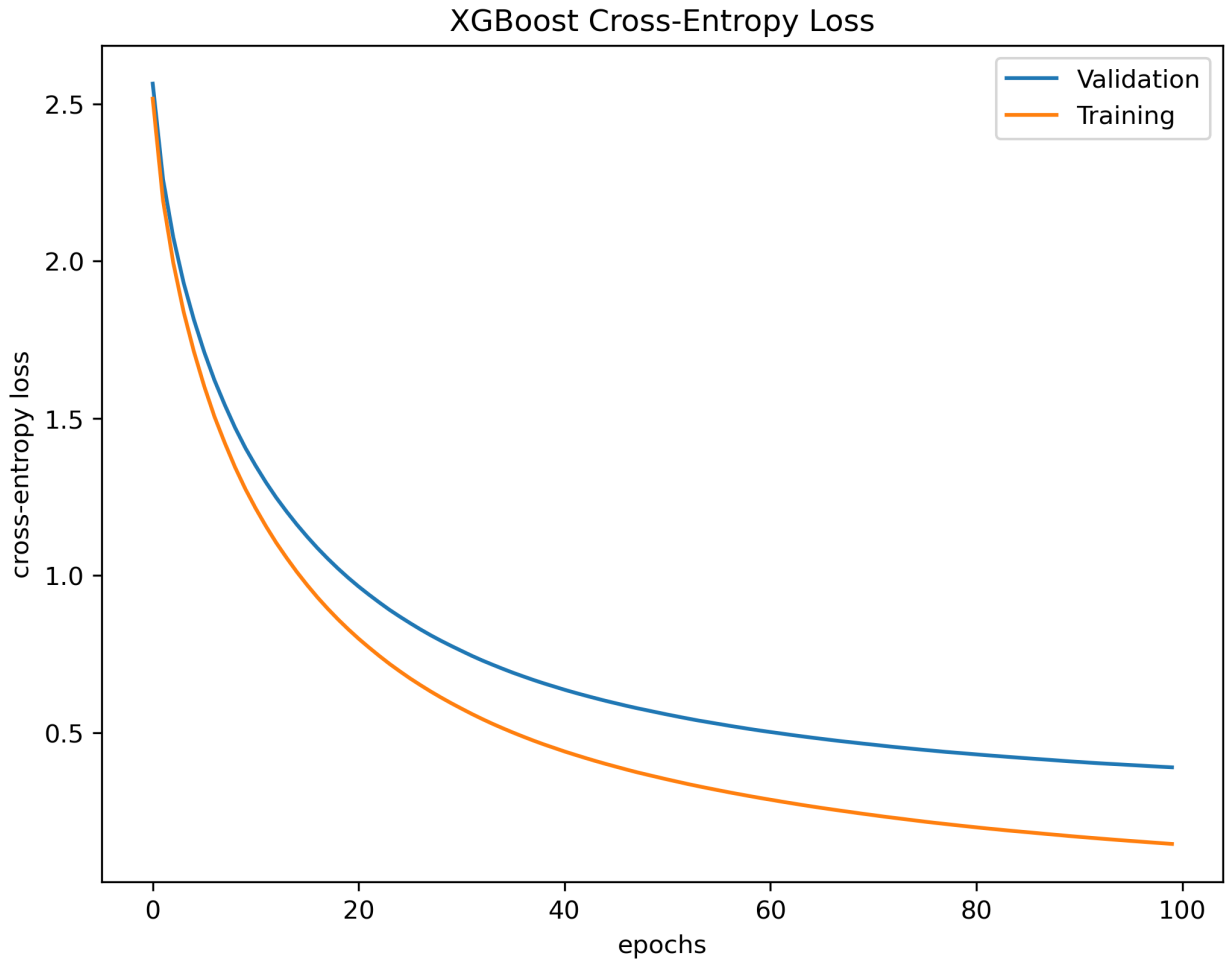


Figure 5.7 This graph visualizes the cross-entropy loss values during each training epoch. We optimize for the validation loss during training. If we observe the loss does not improve over several epochs, we determine that the model has converged and training stops. In this case, our model trained for the entire 100 epochs before hitting this stopping criterion, though the validation loss improvements were marginal in later epochs.

We conclude that the XGBoost model, where each flow is represented by its statistical features, does well classifying our large dataset. It is possible that more aggressive parameter tuning could push these results even higher. We suspect, however, that further tuning would yield only a small increase in the current results, given that we performed moderate tuning already. The tradeoff in time and resources to conduct exhaustive tuning is application dependent and, in

some cases, might be warranted. Second, we note that the performance of the XGBoost model does not exceed that of the transfer learning approach we present in Section 5.3.1. The side-by-side comparison of testing results for both our ResNet50 transfer learning method and the XGBoost method are shown in Figure 5.9.

Table 5.5 This table shows accuracy and f1-scores for the XGBoost model, which uses statistical features for our network traffic flows. We provide training results as well, for completeness.

Dataset	XGBoost Accuracy	XGBoost f1-score
Training	.9571	.9569
Validation	.8852	.8849
Testing	.8920	.8905

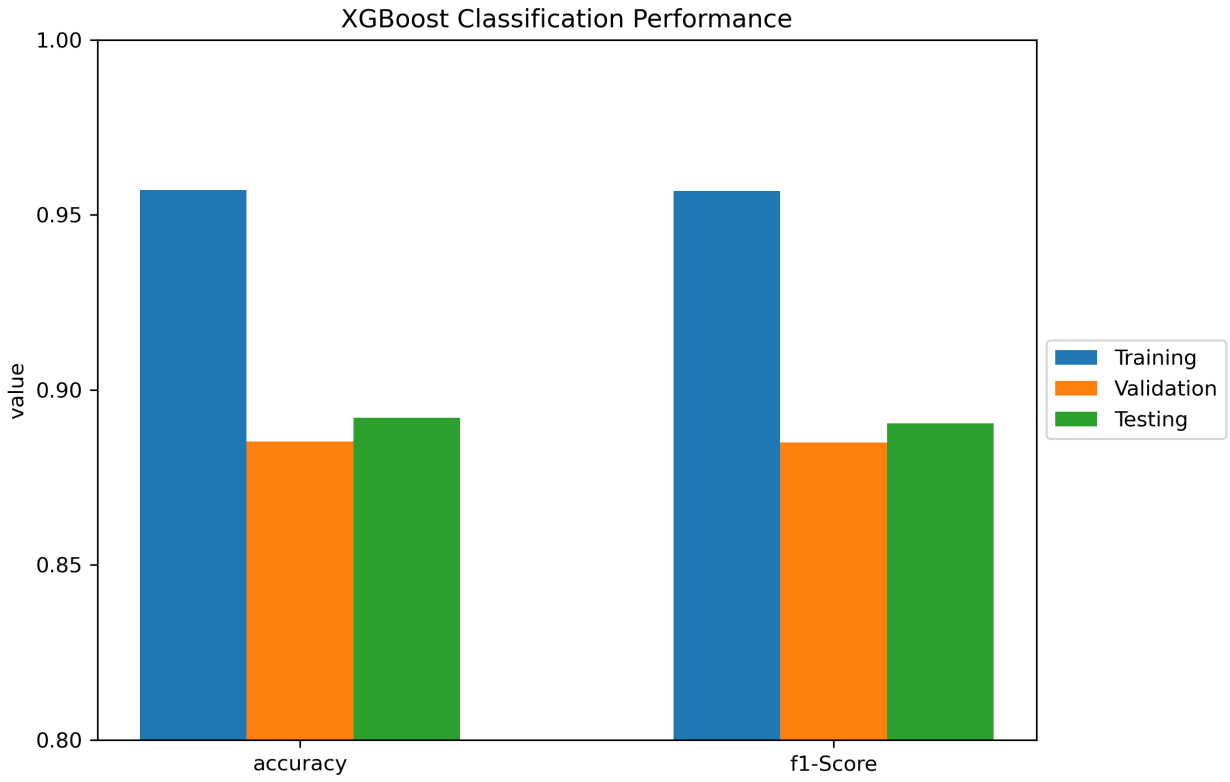


Figure 5.8 This figure shows the accuracy and f1-score results for the XGBoost model. Note that the Y-axis is scaled from 0.8 to 1.0 in order to provide more visual resolution when comparing the results. The XGBoost model performed fairly well when using statistical features to represent each of our network traffic flows. See Table 5.5 for details on the results.

5.5 Related Work

The authors of [8] propose a framework to evaluate several network traffic classification studies and provide valuable lessons learned. For example, they provide much of the insight for representing network traffic flows as bi-flows. Their work helps standardize the research in this field with common language and data pre-processing, which is a first step towards formalizing deep learning for network traffic classification (a task which, as of now, is not as mature as other academic fields where deep learning is applied).

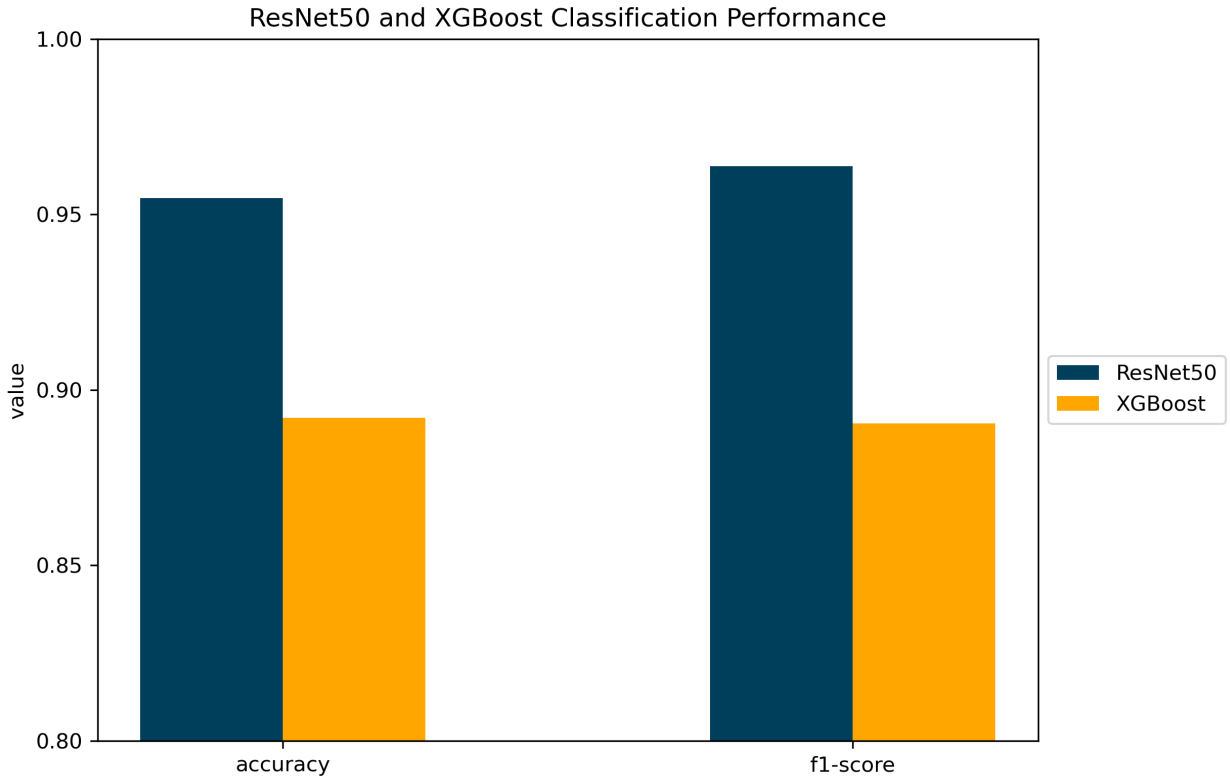


Figure 5.9 This figure shows the head-to-head comparison of the XGBoost model's classification performance on the test set with the fine-tuned ResNet50 model's classification performance. Note that the Y-axis is scaled from 0.8 to 1.0 in order to provide more visual resolution when comparing the results. The XGBoost model classifies flows with calculated statistical features, while the ResNet50 model classifies flows via network traffic images. After adapting to the domain of network traffic classification, ResNet50 has over 5% better classification accuracy than XGBoost for the same traffic flows. See Tables 5.3 and 5.5 for full details on the results of these two techniques.

Recently, some studies have explored representing network traffic as an image. In some cases, as in [65–71], the focus has been on malware. Not all of these studies are using IP packets, as we

do herein, but they still share the same ideas of transforming network traffic objects into images to be classified with CNNs.

A few of these studies [67, 68] use a dataset provided by [66] known as MALIMG. This dataset is far more intricate than what we use in this study. The MALIMG dataset depicts malware in a comprehensive format, with simple packet payloads and timing values as well as the byte code instructions and other domain-specific features. While we find this visualization technique innovative, it is not transferable to the network traffic image classification we conduct in this study, as they rely on the underlying logic of a given type of malware to generate images that differentiate one method from another.

The authors of [69] utilize the IDS2017 traffic dataset and process the data into grayscale images of size 28x28 pixels (784 bytes). Notably, the authors utilize every byte of the IP packet; in our study, we only use the application payload. We feel that only using payload size and timings reveals information about the specific applications rather than capturing other characteristics derived from the operating system or network, such as TCP window sizes. While they discard IP and MAC addressing information, as we do, the inclusion of all other bytes in the IP and TCP or UDP headers makes us question the generalization ability with models trained on such data (i.e., their models are exposed to much more information per packet that is not explicitly a result of the application, but rather the network on which it was captured.)

In [71], the authors also examine malware detection using visualizations of the binary data from packet captures. They utilize a tool known as BINVIS[32] to create RGB images from their replayed packet capture files. The binary representations of their packets include all binary data, not just the application payloads and timings as we do in this study. Notably, this study only uses 1,000 samples during training, which is smaller than any other study we aware of in this domain. They, like us, utilize the ResNet50 model to conduct their training with good results. They observe, however, that this work should be improved by including more training samples.

The authors of [72] create a ResNet50 modification wherein they replace batch normalization with transferable normalization to aid in domain adaptation of ResNet, which was pre-trained on ImageNet, to malware classification. They also convert byte data from the entire packet into an RGB image. Their study utilizes a malware-based and general-use dataset to evaluate their

[32]https://binvis.io//

approach, which has strong accuracy results.

A few examples exist in the recent literature on generic network traffic flows represented as images, specifically in [73–77]. Though the implementations differ from study to study, which we detail in the next few paragraphs, the primary goal of each is to leverage CNNs for two-dimensional traffic images.

In one of the most comparable studies, the authors of [77] conducted experiments that convert network traffic flows to grayscale images and utilize traditional CNN and ResNet models. In this study, the authors did not fine-tune pre-trained models for transfer learning, but still found that ResNet models performed better than traditional CNN models. Additionally, this study only utilized 15 shades of gray, i.e., 4-bit color depth, for each pixel. The entirety of a packet is used to create these images, not just the application payload data (as in our work).

The study presented in [75] focuses on the idea that handcrafted features required for traditional machine learning are cumbersome and expensive. Instead, this work focuses on implementing CNNs with flow sequences converted to images, dubbed Seq2Img in [75]. The authors consider the same packet characteristics that we do in this study, namely payload size, direction, and timing features. However, they further engineer features using Reproducing Kernel Hilbert Space (RKHS) kernel embeddings which we view as complex as the handcrafted features in traditional machine learning. The authors ultimately arrive at 6-channel images generated by a kernel embedding approach that is then fed into a simple CNN. In addition to the image, they also provide the original attributes to the fully connected layer of their CNN, which, in our estimate, further complicates the technique despite the pure image-based model working fairly well.

FlowPic is described in [74]. In this study, the authors turn a network traffic flow into a histogram represented as a 1500x1500 pixel image. The X-axis is the normalized inter-arrival times (from 0-1500) and the Y-axis is the full packet size (including, unlike our work, the Ethernet, IP, and transport layer headers). This work is similar to ours but requires more intermediate engineering steps than we feel are necessary. Additionally, using the entire Ethernet and IP headers makes it more likely to overfit to the training data. They utilized the well-known ISCX-VPN [45] and ISCX-Tor [78] datasets in their study and showed generally competitive results with other previously published techniques using these datasets.

Researchers in [76] describe a grayscale representation of network traffic flows very similar to our implementation. The details on creation are very sparse though, making it hard to compare and contrast their work with ours. While they consider only non-zero payload lengths, as we do, their grayscale image is fed to a 1D-CNN, instead of a 2D-CNN that is typically more well-suited to image recognition. They also use the ISCX-VPN-NonVPN-2016 dataset [45] popular amongst several studies cited in this related works section.

In [73], the authors use grayscale images of 32x32 pixels to represent a packet as we do. They are, however, simply taking the byte representation of their packets and converting them to grayscale images, which works well given that each pixel is represented by 8 bits (0-255), or one byte. They also use a LeNet-5 architecture versus our deeper and more modern ResNet model.

The authors of [46] worked on identifying darknet activity by combining two encrypted traffic datasets. They created an approach called DeepImage, which notably deviates from our technique. DeepImage uses feature selection techniques to extract noteworthy features that are then turned into grayscale images. While this deviates from using simple payload sizes and timing values, it shares commonalities with other works cited in this section. Specifically, using images to represent network traffic is gaining momentum, but the exact features that each pixel represents vary from study to study.

Two works, [3] and [4], explore the application of transfer learning to network traffic classification, though each uses traditional machine learning methods, not deep learning, in their studies. These works help establish that transfer learning is effective and applicable to network traffic classification. We extend prior work to the specific subtask of traffic image classification using deep learning.

A study of deep transfer learning applied to network traffic classification is presented in [79]. Some key differences exist between this prior work and our efforts in this chapter. Specifically, the authors of [79] use statistical features, as we did in Chapters 3 and 4. In contrast, we investigate the impact of using a more raw feature set in this chapter. Furthermore, in [79], the model used as the source to transfer knowledge is not a model trained on known applications, but rather a model using samples from a large unlabeled dataset. The features from the resultant model are then transferred to a new model trained in conjunction with 20 samples from each application. In short, they use a semi-supervised learning technique that addresses the challenge of only a few

labeled instances for applications of interest. For their deep learning model, they utilize a convolutional neural network (CNN), a common architecture employed in deep learning for traffic classification [8], and the basis for the ResNet model we use in this study.

In [80], a recent paper on transfer learning for encrypted traffic classification, the authors utilize a CNN and a Long Short-Term Memory (LSTM) network on the oft-cited ISCX-VPN dataset [45]. The scope of this paper is limited in that they only focus on the Facebook audio, chat, and video classes as the source domain, with the transfer target being Google's hangout chat and audio. This study addresses a key challenge in machine learning: the lack of labeled data.

A few other recent studies focus on implementations of transfer learning for network traffic classification. In [3], the authors examine the ability of traditional machine learning methods to successfully train on data collected at one time and transfer those learned weights to classify data from the same classes collected 12 months later. The authors of [4] similarly explore this topic with a network security emphasis. This study focuses on training with known cyber security threats and transferring those learned weights and parameters to detect new attacks from a target domain.

In summary, several prior works have utilized deep learning, transfer learning from pre-trained models, image representation of network traffic, and other topics presented in our work. Our work, however, is the first to combine all of these techniques together. Representing network traffic in its rawest state is best suited for deep learning, as it allows neural networks to extract the features and weights for classification. Transfer learning poses a significant opportunity for network traffic classification to achieve state-of-the-art results with less overall training time compared with creating fresh models. Finally, while results on standard public datasets are useful for comparison across studies, it's important to evaluate network traffic image classification techniques with real-world datasets, as we have, to understand the suitability for real-world tasks.

5.6 Conclusion and Discussion

Our work in this chapter shows that image classifiers can classify network traffic from large, real-world network captures. Specifically, we effectively transferred the ResNet50 weights pre-trained on ImageNet to our network traffic image classification task, achieving an accuracy score of 95.47% on the test dataset. Furthermore, we describe a technique to convert packet

captures to grayscale images in sufficient detail to be reproduced by other researchers. To provide a point of comparison, we also generated statistical features for the same network traffic flows that we converted to images. We then utilized an XGBoost classifier to learn from these features and attempted to classify the testing dataset. We show that, in this case, the deep learning transfer of ResNet50 to the task of traffic image classification outperforms the traditional machine learning approach with statistical features by a large margin.

Our results showing deep learning is effective for traffic classification is not surprising, as several recent studies have shown good results in this problem space. In fact, visual inspection of the network traffic images often reveals patterns that even humans can detect with the naked eye. What is compelling is the ability to start with a highly trained and renowned computer vision model and adapt it to the traffic classification problem. By morphing raw packet captures into images, essentially an array of 64 bytes reshaped into a two-dimensional 8x8 representation, we can transfer the incredible predictive power of ResNet to our domain.

Several recent studies successfully used one-dimensional CNNs (1D-CNN) to classify network traffic. Our 64-length arrays could be fed to the 1D-CNN without converting to grayscale images in that approach. While this concept is interesting, one main issue is that highly accurate, pre-trained 1D-CNNs are not widely available like ResNet is. Thus, while it would be interesting to compare the two techniques, we must remember that the time and resource savings by transferring learning versus creating models from scratch are incredibly valuable.

Additionally, many studies represent traffic images in an RGB format instead of the grayscale format we adopt in this study. An RGB image is 3-dimensional in that, for each pixel, there are three values representing that pixel's red, green, and blue values. Given that we represent three distinct features in our study (direction, payload size, and inter-arrival timing), it would not be a far stretch to convert our data to RGB to compare how models perform with that format.

Interestingly, two of our top five most confusing classes with the ResNet50 model were FaceTime with iChat and vice versa. Given that iChat is a predecessor to FaceTime, this result makes sense. Still, it also suggests that this technique for traffic image classification can potentially be utilized to generalize to applications of a genre not seen during training. This idea warrants further investigation.

Our initial efforts to determine which of the ResNet models (i.e., ResNet18, ResNet34, ResNet50, etc.) was best suited to the transfer learning task for network traffic image classification was somewhat accelerated. It is possible that excellent results can be had with a simpler version of ResNet than the ResNet50 model we chose for this study. The margin by which we selected ResNet50 was not significantly greater than ResNet18; thus, it is possible that, with diligent hyperparameter tuning and a long enough training time, the ResNet18 results would approach that of the more complex ResNet50.

Finally, the network traffic flows utilized in this study were those observed by our commercial devices within the campus network. That is, none of the traffic flows we used were unidentifiable encrypted data such as SSL or QUIC where the certificates during an initial handshake are not observable, and each had a label given to it. The next logical step for work of this type would be to attempt to apply it to fully labeled, but encrypted, data to determine the ability of traffic image classification to "see through" the encryption and correctly categorize the traffic. Additionally, determining how effective this technique is when encumbered by mostly, or completely, unlabeled data would have a significant real-world impact, as this is the problem usually facing researchers in those spaces. We suspect these are significantly harder tasks but worth investigating.

CHAPTER 6

CONCLUSION

In this dissertation, we examine network traffic classification using machine learning (including deep learning) with the added constraints of having too little labeled data or insufficient time to train models from scratch. Overall, we demonstrate a wide range of applications for machine learning and its potential to solve complex network traffic classification problems when exposed to real-world constraints.

After examining several publicly available datasets for network traffic flows, we determined that, if we wanted to be able to explore applying machine learning models to specific mobile or computer applications, we had to create our own dataset. We reverse-engineered the capture system briefly described in [6] to accomplish this goal. By monitoring the network socket calls made at the operating system level, we were able to demultiplex 100% of the packets from a mobile application on an Android phone. The result was a high-quality, accurate mobile network traffic dataset containing network traffic flows for several encrypted messaging applications not present in any public dataset. Details on our process are in Chapter 2.

Using our custom Android application dataset combined with the public dataset that inspired its creation, we first show in Chapter 3 that utilizing unlabeled data in combination with positively labeled data (positive and unlabeled (PU) learning) is a highly effective technique for network traffic classification using traditional machine learning models (e.g., XGBoost). PU learning provided far better results than other alternatives, such as training models with only the positive data (one-class models) or naively assuming that all of the unlabeled data were negative samples (binary classification).

Our success at classifying an encrypted messaging application, i.e., Signal, in Chapter 3 led us to explore classification performance for this particular genre further. In Chapter 3, one constraint we imposed on our experiments was to have a limited amount of positively labeled data for the target application. In Chapter 4, we take this constraint one step further and provide our PU learning models with zero positive samples for the target class. Specifically, we trained our PU models with samples from a combination of five other encrypted messaging applications that

were not our target. These models were then used to detect our never-before-seen target application, i.e., a zero-day application, with remarkably high accuracy, given that the models had not been trained with data from that target application.

In Chapters 3 and 4, we used our custom Android application dataset combined with the larger public dataset from [6]. Seeking to scale up the size of our dataset and conduct machine learning-based network traffic classification on *real-world data* (i.e., not on a curated dataset), we partnered with our university's information technology services department to obtain a very large network traffic capture from the campus network that spanned an entire week. To address the real-world problem of not having labels for this captured data, we gathered log files from the Palo Alto firewalls in the campus network that were conducting network traffic classification based on information seen during TLS handshakes or by applying heuristics to the network traffic flows. We used these firewall log files to generate *quasi-ground truth* labels for all of the network traffic that we captured from the campus network. Lastly, in Chapters 3 and 4, we utilized tree-based classifiers trained on statistically descriptive features of our network traffic flows. This required pre-calculating all of the features during data capture. For the campus Wi-Fi dataset, we stored the network traffic flow data in a time-series format to better suit the application of deep learning in Chapter 5.

Finally, using this real-world dataset, we examined machine learning for network traffic classification under our second constraint: not wanting or being able to train a model from scratch, possibly due to insufficient time. Rather than starting from a blank slate, we utilized a pre-trained deep learning model as the starting point. Because high-quality pre-trained models for network traffic classification are not widespread today, we transferred a model trained for computer vision tasks. This domain has several state-of-the-art pre-trained convolutional neural networks available. For our work, we chose the ResNet architecture. Since ResNet models are trained to classify images, we first converted our campus network dataset into grayscale images and, as one of the first in the literature to do so, comprehensively described the creation process for other researchers to utilize. With our newly created *network traffic images*, we fine-tuned the ResNet model and achieved over 95% classification accuracy with the testing data. When compared with traditional machine learning tree-based classifiers (i.e., the XGBoost model used in Chapters 3 and 4), our deep transfer learning technique provided a large improvement in model

prediction accuracy.

To simultaneously address both constraints in this dissertation, i.e., not enough labeled data and insufficient time to create and train a fresh model, future work will focus on the deep transfer learning techniques presented in Chapter 5 applied to the problems explored in Chapters 3 and 4. In other words, starting with pre-trained computer vision models, can we effectively classify network traffic images with only a limited amount of positively labeled data? Furthermore, can we fine-tune those pre-trained computer vision models with positively labeled data representing a genre of applications to detect zero-day applications belonging to the same?

In conclusion, our work demonstrates the effectiveness of machine learning for network traffic classification under various constraints. By creating a high-quality, accurate mobile network traffic dataset and leveraging unlabeled data through positive and unlabeled (PU) learning, we show that traditional machine learning models can achieve high accuracy. Furthermore, by utilizing a pre-trained deep learning model and converting network traffic data into grayscale images, we demonstrate the potential of transfer learning in network traffic classification. Our work also highlights the importance of real-world data and the need to address the issue of insufficient labeled data through innovative approaches such as PU learning. Overall, our findings contribute to the growing body of research on machine learning for network traffic classification and provide valuable insights for future work in this area.

REFERENCES

[1] P. Wang, Z. Wang, F. Ye, and X. Chen. ByteSGAN: A Semi-Supervised Generative Adversarial Network for Encrypted Traffic Classification in SDN Edge Gateway. *Computer Networks*, 200:108535, 2021. ISSN 1389-1286. doi: 10.1016/j.comnet.2021.108535. URL https://www.sciencedirect.com/science/article/pii/S138912862100459X.

[2] A.S. Iliyasu and H. Deng. Semi-Supervised Encrypted Traffic Classification With Deep Convolutional Generative Adversarial Networks. *IEEE Access*, 8:118–126, 2020. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2962106.

[3] G. Sun, L. Liang, T. Chen, F. Xiao, and F. Lang. Network Traffic Classification Based on Transfer Learning. *Computers & Electrical Engineering*, 69:920–927, 2018. ISSN 0045-7906. doi: 10.1016/j.compeleceng.2018.03.005.

[4] J. Zhao, S. Shetty, and J.W. Pan. Feature-Based Transfer Learning for Network Security. In *MILCOM 2017 - 2017 IEEE Military Communications Conference (MILCOM)*, pages 17–22, 2017.

[5] J. Hussey, E. Taylor, K. Stone, and T. Camp. Poster: Data Collection for ML Classification of Encrypted Messaging Applications. In *2021 IEEE 29th International Conference on Network Protocols (ICNP)*, pages 1–2, 2021. doi: 10.1109/ICNP52444.2021.9651948. ISSN: 2643-3303.

[6] G. Aceto, D. Ciuonzo, A. Montieri, V. Persico, and A. Pescape. MIRAGE: Mobile-App Traffic Capture and Ground-Truth Creation. In *2019 4th International Conference on Computing, Communications and Security (ICCCS)*, pages 1–8, 2019.

[7] L. Yang, A. Finamore, F. Jun, and D. Rossi. Deep Learning and Zero-Day Traffic Classification: Lessons Learned From a Commercial-Grade Dataset. *IEEE Transactions on Network and Service Management*, 18(4):4103–4118, 2021. ISSN 1932-4537. doi: 10.1109/TNSM.2021.3122940.

[8] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé. Mobile Encrypted Traffic Classification Using Deep Learning: Experimental Evaluation, Lessons Learned, and Challenges. *IEEE Transactions on Network and Service Management*, 16(2):445–458, 2019. ISSN 1932-4537. doi: 10.1109/TNSM.2019.2899085.

[9] J. Hussey, K. Stone, and T. Camp. Positive and Unlabeled Learning for Mobile Application Traffic Classification. In *MILCOM 2022 - 2022 IEEE Military Communications Conference (MILCOM)*, pages 25–30, 2022. doi: 10.1109/MILCOM55135.2022.10017699. ISSN: 2155-7586.

[10] T.T.T. Nguyen and G. Armitage. A Survey of Techniques for Internet Traffic Classification Using Machine Learning. *IEEE Communications Surveys Tutorials*, 10(4):56–76, 2008. ISSN 1553-877X. doi: 10.1109/SURV.2008.080406.

[11] E. Papadogiannaki and S. Ioannidis. A Survey on Encrypted Network Traffic Analysis Applications, Techniques, and Countermeasures. *ACM Computing Surveys*, 54(6):1–35, 2021. ISSN 0360-0300, 1557-7341. doi: 10.1145/3457904.

[12] A. Dainotti, A. Pescape, and K.C. Claffy. Issues and Future Directions in Traffic Classification. *IEEE Network*, 26(1):35–40, 2012. ISSN 1558-156X. doi: 10.1109/MNET.2012.6135854.

[13] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. BLINC: Multilevel Traffic Classification in the Dark. *ACM SIGCOMM Computer Communication Review*, 35(4):229–240, 2005. ISSN 0146-4833. doi: 10.1145/1090191.1080119. URL https://doi.org/10.1145/1090191.1080119.

[14] C. Elkan and K. Noto. Learning Classifiers From Only Positive and Unlabeled Data. In *14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD 08*, pages 213–220, 2008. ISBN 978-1-60558-193-4. doi: 10.1145/1401890.1401920.

[15] F. Mordelet and J-P. Vert. A Bagging SVM to Learn From Positive and Unlabeled Examples. *Pattern Recognition Letters*, 2014.

[16] J. Bekker and J. Davis. Learning From Positive and Unlabeled Data: A Survey. *Machine Learning*, 109:719–760, 2020. ISSN 1573-0565. doi: 10.1007/s10994-020-05877-5.

[17] A. Kaboutari, J. Bagherzadeh, and F. Kheradmand. An Evaluation of Two-Step Techniques for Positive-Unlabeled Learning in Text Classification. *International Journal of Computer Applications Technology and Research*, 3(9):3, 2014.

[18] B. Wang, K. Yu, X. Wu, F. Wei, W. Jiang, and D. Pan. Positive and Unlabeled Learning for Mobile App User and Server Interaction Prediction. In B. Li, L. Shu, and D. Zeng, editors, *Communications and Networking*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 481–491, 2018. ISBN 978-3-319-78130-3. doi: 10.1007/978-3-319-78130-3_50.

[19] A. Este, F. Gringoli, and L. Salgarelli. Support Vector Machines for TCP Traffic Classification. *Computer Networks*, 53(14):2476–2490, 2009. ISSN 13891286. doi: 10.1016/j.comnet.2009.05.003.

[20] Y. Lim, H. Kim, J. Jeong, C. Kim, T. Kwon, and Y. Choi. Internet Traffic Classification Demystified: On The Sources of The Discriminative Power. In *6th International Conference*, Co-NEXT '10, pages 1–12, 2010. ISBN 978-1-4503-0448-1. doi: 10.1145/1921168.1921180.

[21] Y. Fu, H. Xiong, X. Lu, J. Yang, and C. Chen. Service Usage Classification with Encrypted Internet Traffic in Mobile Messaging Apps. *IEEE Transactions on Mobile Computing*, 15 (11):2851–2864, 2016. ISSN 1558-0660. doi: 10.1109/TMC.2016.2516020.

[22] V. F. Taylor, R. Spolaor, M. Conti, and I. Martinovic. Robust Smartphone App Identification via Encrypted Network Traffic Analysis. *IEEE Transactions on Information Forensics and Security*, 13(1):63–78, 2018. ISSN 1556-6021. doi: 10.1109/TIFS.2017.2737970.

[23] S.E. Coull and K.P. Dyer. Traffic Analysis of Encrypted Messaging Services: Apple iMessage and Beyond. *ACM SIGCOMM Computer Communication Review*, 44(5):6, 2014.

[24] I.L. Cherif and A. Kortebi. On Using eXtreme Gradient Boosting (XGBoost) Machine Learning Algorithm for Home Network Traffic Classification. In *2019 Wireless Days (WD)*, pages 1–6, 2019. doi: 10.1109/WD.2019.8734193.

[25] T. Chen and C. Guestrin. XGBoost: A Scalable Tree Boosting System. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2016. ISBN 978-1-4503-4232-2. doi: 10.1145/2939672.2939785.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] B. Schölkopf, R.C. Williamson, A. Smola, J. Shawe-Taylor, and J. Platt. Support Vector Method for Novelty Detection. In *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 2000.

[28] F.T. Liu, K.M. Ting, and Z.-H. Zhou. Isolation Forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422, 2008. doi: 10.1109/ICDM.2008.17.

[29] F.T. Liu, K.M. Ting, and Z. Zhou. Isolation-Based Anomaly Detection. *ACM Transactions on Knowledge Discovery from Data*, 6(1):3:1–3:39, 2012. ISSN 1556-4681. doi: 10.1145/2133360.2133363.

[30] N. Fu, Y. Xu, J. Zhang, R. Wang, and J. Xu. FlowCop: Detecting 'Stranger' in Network Traffic Classification. In *2018 27th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9, 2018. doi: 10.1109/ICCCN.2018.8487398.

[31] V. F. Taylor, R. Spolaor, M. Conti, and .I Martinovic. AppScanner: Automatic Fingerprinting of Smartphone Apps from Encrypted Network Traffic. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 439–454, 2016. doi: 10.1109/EuroSP.2016.40.

[32] A. Callado, C. Kamienski, G. Szabo, B.P. Gero, J. Kelner, S. Fernandes, and D. Sadok. A Survey on Internet Traffic Identification. *IEEE Communications Surveys Tutorials*, 11(3): 37–52, 2009. ISSN 1553-877X. doi: 10.1109/SURV.2009.090304.

[33] M. Conti, Q.Q. Li, A. Maragno, and R. Spolaor. The Dark Side(-Channel) of Mobile Devices: A Survey on Network Traffic Analysis. *IEEE Communications Surveys Tutorials*, 20 (4):2658–2713, 2018. ISSN 1553-877X. doi: 10.1109/COMST.2018.2843533.

[34] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé. Multi-classification Approaches for Classifying Mobile App Traffic. *Journal of Network and Computer Applications*, 103: 131–145, 2018. ISSN 1084-8045. doi: 10.1016/j.jnca.2017.11.007. URL https://www.sciencedirect.com/science/article/pii/S1084804517303740.

[35] G. Aceto, A. Dainotti, W. de Donato, and A. Pescape. PortLoad: Taking the Best of Two Worlds in Traffic Classification. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–5, 2010. doi: 10.1109/INFCOMW.2010.5466645.

[36] J. Li, H. Zhou, S. Wu, X. Luo, T. Wang, X. Zhan, and X. Ma. FOAP: Fine-Grained Open-World Android App Fingerprinting. *31st USENIX Security Symposium*, 2022.

[37] J. Zhang, Z. Wang, J. Yuan, and Y.-P. Tan. Positive and Unlabeled Learning for Anomaly Detection with Multi-features. In *25th ACM International Conference on Multimedia*, MM '17, pages 854–862, 2017. ISBN 978-1-4503-4906-2. doi: 10.1145/3123266.3123304. URL https://doi.org/10.1145/3123266.3123304.

[38] K. Yu, Y. Liu, L. Qing, B. Wang, and Y. Cheng. Positive and Unlabeled Learning for User Behavior Analysis Based on Mobile Internet Traffic Data. *IEEE Access*, 6:37568–37580, 2018. ISSN 2169-3536. doi: 10.1109/ACCESS.2018.2852008.

[39] R. Wright. Positive-Unlabeled Learning, 2017. URL https://roywrightme.wordpress.com/2017/11/16/positive-unlabeled-learning/.

[40] F. Meslet-Millet, E. Chaput, and S. Mouysset. SPPNet: An Approach For Real-Time Encrypted Traffic Classification Using Deep Learning. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021. doi: 10.1109/GLOBECOM46510.2021.9686037.

[41] G. Aceto, D. Ciuonzo, A. Montieri, and A. Pescapé. DISTILLER: Encrypted Traffic Classification via Multimodal Multitask Deep Learning. *Journal of Network and Computer Applications*, 183-184:102985, 2021. ISSN 10848045. doi: 10.1016/j.jnca.2021.102985. URL https://linkinghub.elsevier.com/retrieve/pii/S1084804521000126.

[42] S. Rezaei and X. Liu. Multitask Learning for Network Traffic Classification. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9, 2020. doi: 10.1109/ICCCN49398.2020.9209652. ISSN: 2637-9430.

[43] W. Wang, M. Zhu, J. Wang, X. Zeng, and Z. Yang. End-to-End Encrypted Traffic Classification with One-Dimensional Convolution Neural Networks. In *2017 IEEE International Conference on Intelligence and Security Informatics (ISI)*, pages 43–48, 2017. doi: 10.1109/ISI.2017.8004872.

[44] Y. Huo, C. Song, M. Zhou, R. Lv, and Y. Yang. A Novel Approach for Semi-Supervised Network Traffic Classification. In *2022 IEEE 14th International Conference on Advanced Infocomm Technology (ICAIT)*, pages 64–69, 2022. doi: 10.1109/ICAIT56197.2022.9862675. ISSN: 2770-1603.

[45] G. Draper-Gil, A.H. Lashkari, M.S.I. Mamun, and A. A. Ghorbani. Characterization of Encrypted and VPN Traffic using Time-related Features:. In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy*, pages 407–414, 2016. ISBN 978-989-758-167-0. doi: 10.5220/0005740704070414.

[46] A. Habibi Lashkari, G. Kaur, and A. Rahali. DIDarknet: A Contemporary Approach to Detect and Characterize the Darknet Traffic using Deep Image Learning. In *2020 the 10th International Conference on Communication and Network Security*, ICCNS 2020, pages 1–13, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 978-1-4503-8903-7. doi: 10.1145/3442520.3442521. URL https://doi.org/10.1145/3442520.3442521.

[47] Y. Li, Y. Lu, and S. Li. EZAC: Encrypted Zero-day Applications Classification using CNN and K-Means. In *2021 IEEE 24th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 378–383, 2021. doi: 10.1109/CSCWD49262.2021.9437716.

[48] Y. Gu, D. Li, K. Gao, and Y. Cheng. Fast and Robust Online Traffic Classification Supporting Unseen Applications. In *2021 IEEE Global Communications Conference (GLOBECOM)*, pages 1–6, 2021. doi: 10.1109/GLOBECOM46510.2021.9685631.

[49] J. Zhang, X. Chen, Y. Xiang, W. Zhou, and Jie Wu. Robust Network Traffic Classification. *IEEE/ACM Transactions on Networking*, 23(4):1257–1270, 2015. ISSN 1558-2566. doi: 10.1109/TNET.2014.2320577.

[50] J. Zhang, X. Chen, Y. Xiang, and W. Zhou. Zero-Day Traffic Identification. In G. Wang, I. Ray, D. Feng, and M. Rajarajan, editors, *Cyberspace Safety and Security*, pages 213–227, Cham, 2013. ISBN 978-3-319-03584-0. doi: 10.1007/978-3-319-03584-0_16.

[51] M. Shen, Y. Liu, L. Zhu, K. Xu, X. Du, and N. Guizani. Optimizing Feature Selection for Efficient Encrypted Traffic Classification: A Systematic Approach. 34(4):20–27, 2020. ISSN 1558-156X. doi: 10.1109/MNET.011.1900366.

[52] S. Niu, Y. Liu, J. Wang, and H. Song. A Decade Survey of Transfer Learning (2010–2020). 1 (2):151–166, 2020. ISSN 2691-4581. doi: 10.1109/TAI.2021.3054609.

[53] P. Wang, X. Chen, F. Ye, and Z. Sun. A Survey of Techniques for Mobile Service Encrypted Traffic Classification Using Deep Learning. 7:54024–54033, 2019. ISSN 2169-3536. doi: 10.1109/ACCESS.2019.2912896.

[54] P. Velan, M. Čermák, P. Čeleda, and M. Drašar. A Survey of Methods for Encrypted Traffic Classification and Analysis. *Networks*, 25(5):355–374, 2015. ISSN 0028-3045.

[55] L.A. Iliadis and T. Kaifas. Darknet Traffic Classification using Machine Learning Techniques. In *2021 10th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, pages 1–4, 2021. doi: 10.1109/MOCAST52088.2021.9493386.

[56] R. McKay, B. Pendleton, J. Britt, and B. Nakhavanit. Machine Learning Algorithms on Botnet Traffic: Ensemble and Simple Algorithms. In *Proceedings of the 2019 3rd International Conference on Compute and Data Analysis*, ICCDA 2019, pages 31–35, 2019. ISBN 978-1-4503-6634-2. doi: 10.1145/3314545.3314569. URL https://doi.org/10.1145/3314545.3314569.

[57] L. Bernaille, R. Teixeira, I. Akodkenou, A. Soule, and K. Salamatian. Traffic Classification On The Fly. *ACM SIGCOMM Computer Communication Review*, 36(2):23–26, 2006. ISSN 0146-4833. doi: 10.1145/1129582.1129589. URL https://dl.acm.org/doi/10.1145/1129582.1129589.

[58] K. He, X. Zhang, S. Ren, and J. Sun. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016. doi: 10.1109/CVPR.2016.90. ISSN: 1063-6919.

[59] J. Howard. fastai, 2018. URL https://github.com/fastai/fastai.

[60] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009. doi: 10.1109/CVPR.2009.5206848.

[61] K.P. Dyer, S.E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail. In *2012 IEEE Symposium on Security and Privacy*, pages 332–346, 2012. ISBN 978-1-4673-1244-8 978-0-7695-4681-0. doi: 10.1109/SP.2012.28. URL http://ieeexplore.ieee.org/document/6234422/.

[62] M.D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks, 2013. URL http://arxiv.org/abs/1311.2901.

[63] H. Zhang, M. Cisse, Y.N. Dauphin, and D. Lopez-Paz. mixup: Beyond Empirical Risk Minimization, 2018. URL http://arxiv.org/abs/1710.09412.

[64] N. Manju, B.S. Harish, and V. Prajwal. Ensemble Feature Selection and Classification of Internet Traffic using XGBoost Classifier. *International Journal of Computer Network and Information Security*, 11(7):37–44, 2019. ISSN 20749090, 20749104. doi: 10.5815/ijcnis.2019.07.06. URL http://www.mecs-press.org/ijcnis/ijcnis-v11-n7/v11n7-6.html.

[65] W. Wang, M. Zhu, X. Zeng, X. Ye, and Y. Sheng. Malware traffic classification using convolutional neural network for representation learning. In *2017 International Conference on Information Networking (ICOIN)*, pages 712–717, 2017. doi: 10.1109/ICOIN.2017.7899588.

[66] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th International Symposium on Visualization for Cyber Security*, pages 1–7, Pittsburgh Pennsylvania USA, 2011. ACM. ISBN 978-1-4503-0679-9. doi: 10.1145/2016904.2016908. URL https://dl.acm.org/doi/10.1145/2016904.2016908.

[67] D. Vasan, M. Alazab, S. Wassan, B. Safaei, and Q. Zheng. Image-Based malware classification using ensemble of CNN architectures (IMCEC). *Computers & Security*, 92: 101748, 2020. ISSN 0167-4048. doi: 10.1016/j.cose.2020.101748. URL https://www.sciencedirect.com/science/article/pii/S016740482030033X.

[68] D. Pant and R. Bista. Image-based Malware Classification using Deep Convolutional Neural Network and Transfer Learning. In *Proceedings of the 3rd International Conference on Advanced Information Science and System*, AISS '21, pages 1–6. Association for Computing Machinery, 2022. ISBN 978-1-4503-8586-2. doi: 10.1145/3503047.3503081. URL https://dl.acm.org/doi/10.1145/3503047.3503081.

[69] G. Agrafiotis, E. Makri, I. Flionis, A. Lalas, K. Votis, and D. Tzovaras. Image-based Neural Network Models for Malware Traffic Classification using PCAP to Picture Conversion. In *Proceedings of the 17th International Conference on Availability, Reliability and Security*, ARES '22, pages 1–7. Association for Computing Machinery, 2022. ISBN 978-1-4503-9670-7. doi: 10.1145/3538969.3544473. URL https://dl.acm.org/doi/10.1145/3538969.3544473.

[70] D. Vasan, M. Alazab, S. Wassan, H. Naeem, B. Safaei, and Q. Zheng. IMCFN: Image-based malware classification using fine-tuned convolutional neural network architecture. *Computer Networks*, 171:107–138, 2020. ISSN 13891286. doi: 10.1016/j.comnet.2020.107138. URL https://linkinghub.elsevier.com/retrieve/pii/S1389128619304736.

[71] G. Bendiab, S. Shiaeles, A. Alruban, and N. Kolokotronis. IoT Malware Network Traffic Classification using Visual Representation and Deep Learning. In *2020 6th IEEE Conference on Network Softwarization (NetSoft)*, pages 444–449, 2020. doi: 10.1109/NetSoft48620.2020.9165381.

[72] C. Rong, G. Gou, M. Cui, G. Xiong, Z. Li, and L. Guo. TransNet: Unseen Malware Variants Detection Using Deep Transfer Learning. In *Security and Privacy in Communication Networks*, Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, pages 84–101, Cham, 2020. Springer International Publishing. ISBN 978-3-030-63095-9. doi: 10.1007/978-3-030-63095-9_5.

[73] Y. Zhou, H. Shi, Y. Zhao, W. Gao, and W. Zhang. Encrypted Network Traffic Identification Based on 2D-CNN Model. In *2021 22nd Asia-Pacific Network Operations and Management Symposium (APNOMS)*, pages 238–241, 2021. doi: 10.23919/APNOMS52696.2021.9562636.

[74] T. Shapira and Y. Shavitt. FlowPic: Encrypted Internet Traffic Classification is as Easy as Image Recognition. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 680–687, 2019. doi: 10.1109/INFCOMW.2019.8845315.

[75] Z. Chen, K. He, J. Li, and Y. Geng. Seq2Img: A sequence-to-image based approach towards IP traffic classification using convolutional neural networks. In *2017 IEEE International Conference on Big Data (Big Data)*, pages 1271–1276, 2017. doi: 10.1109/BigData.2017.8258054.

[76] Y. He and W. Li. Image-based Encrypted Traffic Classification with Convolution Neural Networks. In *2020 IEEE Fifth International Conference on Data Science in Cyberspace (DSC)*, pages 271–278, 2020. doi: 10.1109/DSC50466.2020.00048.

[77] H.-K. Lim, J.-B. Kim, J.-S. Heo, K. Kim, Y.-G. Hong, and Y.-H. Han. Packet-based Network Traffic Classification Using Deep Learning. In *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)*, pages 046–051, 2019. doi: 10.1109/ICAIIC.2019.8669045.

[78] A. Habibi Lashkari, G. Draper Gil, M.S. Mamun, and A.A. Ghorbani. Characterization of Tor Traffic using Time based Features:. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy*, pages 253–262. SCITEPRESS - Science and Technology Publications, 2017. ISBN 978-989-758-209-7. doi: 10.5220/0006105602530262.

[79] S. Rezaei and X. Liu. How to Achieve High Classification Accuracy with Just a Few Labels: A Semi-supervised Approach Using Sampled Packets. *arXiv:1812.09761 [cs]*, 2020. URL http://arxiv.org/abs/1812.09761. arXiv: 1812.09761.

[80] X. Hu, C. Gu, Y. Chen, and F. Wei. tCLD-Net: A Transfer Learning Internet Encrypted Traffic Classification Scheme Based on Convolution Neural Network and Long Short-Term Memory Network. In *2021 International Conference on Communications, Computing, Cybersecurity, and Informatics (CCCI)*, pages 1–5, 2021. doi: 10.1109/CCCI52664.2021.9583214.

APPENDIX A

COPYRIGHT PERMISSIONS

Requirements to be followed when using any portion (e.g., figure, graph, table, or textual material) of an IEEE copyrighted paper in a thesis:

1. In the case of textual material (e.g., using short quotes or referring to the work within these papers) users must give full credit to the original source (author, paper, publication) followed by the IEEE copyright line © 2011 IEEE.

2. In the case of illustrations or tabular material, we require that the copyright line © [Year of original publication] IEEE appear prominently with each reprinted figure and/or table.

3. If a substantial portion of the original paper is to be used, and if you are not the senior author, also obtain the senior author's approval.

Requirements to be followed when using an entire IEEE copyrighted paper in a thesis:

1. The following IEEE copyright/ credit notice should be placed prominently in the references: © [year of original publication] IEEE. Reprinted, with permission, from [author names, paper title, IEEE publication title, and month/year of publication]

2. Only the accepted version of an IEEE copyrighted paper can be used when posting the paper or your thesis on-line.

3. In placing the thesis on the author's university website, please display the following message in a prominent place on the website: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [university/educational entity's name goes here]'s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to

http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink.

If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

APPENDIX B

COAUTHOR PERMISSIONS

Permission to include portions of our publication [5] in Chapter 2 as part of this thesis (obtained from E. Taylor, who is not on the thesis committee) is expressed below:

Re: Request to use our poster in Ph.D. Thesis

Ethan Taylor (Student) <ewtaylor@mines.edu>
Tue 5/2/2023 11:40 AM
To: Jason Hussey (Student) <jasonhussey@mines.edu>

Jason,

Yes, I am willing to grant reuse permission for the poster.

Best,
Ethan Taylor

From: Jason Hussey (Student) <jasonhussey@mines.edu>
Sent: Tuesday, May 2, 2023 11:19 AM
To: Ethan Taylor (Student) <ewtaylor@mines.edu>
Subject: Request to use our poster in Ph.D. Thesis

Hi Ethan,

I would like to use portions of our published poster, "Poster: Data Collection for ML Classification of Encrypted Messaging Applications" (https://doi.org/10.1109/ICNP52444.2021.9651948), in a chapter for my Ph.D. dissertation. Would you be willing to grant reuse permissions for this?

Respectfully,
Jason Hussey

Ph.D. Student
Computer Science
Colorado School of Mines