Towards an End-to-End System for Threat Detection on Enterprise Networks

---

A Thesis

Presented to

the faculty of the School of Engineering and Applied Science

University of Virginia

---

in partial fulfillment
of the requirements for the degree

Master of Science

by

Brendan Abraham

May 2019

APPROVAL SHEET

This Thesis
is submitted in partial fulfillment of the requirements
for the degree of
Master of Science

Author Signature: Brendan Abraham

This Thesis has been read and approved by the examining committee:

Advisor: Dr. Donald Brown (Co-advisor)

Committee Member: Dr. Veeraraghavan (Co-advisor)

Committee Member: Dr. Matthew Gerber

Committee Member: _____

Committee Member: _____

Committee Member: _____

Accepted for the School of Engineering and Applied Science:

Craig H. Benson, School of Engineering and Applied Science

May 2019

UNIVERSITY OF VIRGINIA

Towards an End-to-End System for Threat Detection on Enterprise Networks

---

Author:

Brendan ABRAHAM

Advisors:

Dr. Don BROWN

Dr. Malathi Veeraraghavan

A thesis submitted in fulfillment of the requirements
for the degree of Master of Science

in the

Department or Systems and Information Engineering

April 23, 2019

Abstract

Cybercrime has become one of the most pressing issues of the digital age. Cyber-attacks cost businesses on average $11 million annually and there are no signs of slowing down, as the amount of attacks has doubled in the last five years [13]. Moreover, attackers are employing increasingly sophisticated methods, targeting government, business, and academic institutions alike. The traditional defense against cybercrime is an Intrusion Detection System (IDS) which examines network traffic for malicious or anomalous behavior. In most cases, these systems use signature-based detection, relying on pre-defined rules and attack signatures to detect intrusions. This strategy is woefully inadequate in an age where cyber-attacks are constantly evolving. A more promising approach is anomaly detection, which models host or endpoint behavior on a network in order to detect malicious traffic. In these systems, predictive models are trained to learn behavioral patterns from the data, as opposed to being told exactly what to look for.

In this thesis, we lay the groundwork for an end-to-end, anomaly-based Intrusion Detection System for UVA network traffic. Our work consists of three components. First, we demonstrate through a pilot study that machine learning techniques can be extremely effective at isolating botnet traffic and potentially detecting zero-day attacks. We present a novel evaluation cross-validation technique called Leave-One-Bot-Out CV (LOBO-CV) which effectively measures a model's ability to generalize to traffic from a new, unseen botnet. Second, we present a high-speed traffic capturing pipeline and apply it to our own network data. Finally, we present a traffic labeling pipeline leveraging blacklist, whitelist, and honeypot feeds to label our network traffic on a daily basis. Experimental results suggest that the labels produced by this pipeline are legitimate and that malicious traffic can be isolated from whitelisted traffic if the right features and model are used.

Acknowledgements

I would like to take this opportunity to acknowledge and thank the people that helped me and worked with me throughout my graduate school career. Without their support, this thesis would not have been possible.

I would like to thank my Systems advisor, Professor Don Brown. His insights, support, and guidance were extremely helpful in not only crafting my thesis, but also helping me get through graduate school. It was simultaneously humbling and a pleasure to work for someone as accomplished and knowledgeable in Data Science as he is.

I would also like to thank Professor Malathi Veeraraghavan, my co-advisor from the Electrical and Computer Engineering department. She taught me everything I know about computer networks and many important lessons ranging from how to diagnose a bug in C code to how to write an effective email. Most of all, she taught me the importance of developing a deep understanding of the problem at hand before delving into solutions. I'd also like to thank her for her guidance in my research and feedback on my thesis.

Additionally, I would like to thank Professor Gerber for his invaluable feedback on my thesis and defense presentation.

The work presented in this thesis was highly collaborative. As such, I'd like to take this time to thank all the people who worked with me on each project and acknowledge their contributions.

First, I'd like to thank Rohan Bapat, Abhijith Mandya and Fatma Al-Ali for their contributions to the botnet detection study. Abhijith wrote code for many of the individual models, which I simply executed and extended to build the remaining models. Rohan was instrumental in implementing our 'Leave-One-Bot-Out' cross-validation scheme. Last, but certainly not least, Fatma pointed us to the dataset we used for this project in the first place. Without her research and assistance, the project would not have been possible.

Additionally, I'd like to thank the people involved in building our traffic collection pipeline. First, I'd like to thank Jeff Collyer for configuring the Gigamon and for all his help with debugging packet loss issues. Second, I'd like to thank Sourav

Maji for all his guidance and assistance in setting up the pipeline. His help was invaluable in configuring Ivy Bulwark's environment to work with Bro and some of the key optimizations presented herein were originally his ideas (noted in chapter 4). Third, I'd like to thank Jack Morris for his help debugging issues with Bro and for building a performance dashboard. Fourth, I'd like to thank Curtis Kahn for automating data transfer in the pipeline and for setting up log anonymization. Finally, I'd like to thank Alex Ptak and Michelle Co for their help with administrative issues on Ivy Bulwark. Without the contributions of these people, completing this project alone would have been next to impossible.

Moreover, I'd like to thank the people that worked with me on building the traffic labeling system. This includes Yizhe Zhang for developing many of the blacklist crawlers, Alastair Nottingham for helping me design the initial database schema, and Will Hawkins for his feedback on my code. Additionally, I'd like to thank the 2018 MSDS Cyber Capstone team, namely Rakesh Ravi, Boda Ye, and David Roden, for their work on clustering labeled traffic. While reproduced on a different sample, the results presented in 5.3 are based off of their methods.

Finally, I would like to thank my parents for their moral support throughout graduate school.

It was a pleasure working with everyone over the last two years. Together, we accomplished significantly more than I could have possibly done on my own.

This research was funded by SEAS RIA and the DARPA PCORE project.

Contents

Abstract **i**

Acknowledgements **ii**

1 Introduction **1**

1.1 Introduction 1

1.2 Objectives 2

1.3 Contributions 2

1.4 Thesis Organization 2

2 Background & Related Work **3**

2.1 Networking Concepts & Terminology 3

2.2 Intrusion Detection Systems 5

2.3 Machine Learning Approaches to Threat Detection 5

2.4 High Speed Packet Capture Solutions 7

2.5 Approaches to Labeling Network Traffic 8

3 Evaluating Machine Learning Approaches for Botnet Detection **11**

3.1 Introduction 11

3.2 Background 12

3.3 Objectives 13

3.4 Data Description 13

3.5 Data Processing 14

3.6 Feature Engineering 16

3.7 Modeling Overview 17

3.8 Modeling Techniques 18

3.8.1 Logistic Regression 18

3.8.2 Naive Bayes . . . 19
3.8.3 Support Vector Machines (Radial) . . . 19
3.8.4 Random Forest . . . 20
3.8.5 Fully-Connected Neural Network . . . 20
3.9 Evaluation and Results . . . 21
3.9.1 K-fold CV . . . 21
3.9.2 LOBO CV . . . 22
3.9.3 Ensemble Results . . . 23
3.10 Conclusions and Applications to UVA Data . . . 24

4 UVA Traffic Processing Pipeline 26
4.1 System Overview . . . 27
4.2 Packet Forwarding with the Gigamon . . . 28
4.3 Packet Processing . . . 28
4.3.1 Processing Steps . . . 29
4.3.2 Processing Challenges . . . 29
4.3.3 Metrics . . . 30
Packet Loss (Packet Drop Rate) . . . 30
CaptureLoss . . . 30
Link Utilization . . . 31
4.3.4 Optimizations . . . 31
Increasing the Buffer Size (NIC) . . . 32
Load Balancing (Bro) . . . 32
Minimal Log Writing (Bro) . . . 34
Subnet Filtering (Gigamon) . . . 34
Avoiding Memory Leaks with Daily Restarts (Bro) . . . 35
Disabling RX/TX Offloading (NIC) . . . 37
Disabling Checksum Validation (Bro) . . . 37
4.3.5 Summary . . . 38
4.4 Log Anonymization . . . 39
4.5 Conclusion . . . 39

5 UVA Traffic Labeling Pipeline **41**

5.1 Labeling System Architecture 41

5.1.1 Design Considerations 47

5.2 Implementing the System for Our Research 48

5.2.1 Categories 48

Honeypots 48

Blacklists 49

Whitelists 49

5.2.2 Feeds 49

5.3 Exploratory Analysis 50

5.3.1 Cluster Analysis 51

5.3.2 Port Analysis 52

5.3.3 Geolocation Analysis 52

5.3.4 Conclusions 53

5.4 Future Improvements 53

6 Conclusions and Future Work **54**

6.1 Conclusions 54

6.2 Limitations and Future Work 56

Bibliography **58**

List of Figures

2.1 An illustration of different traces. Graphic courtesy of Botfinder [22] . 4

3.1 An illustration of our neural network. 16
3.2 An illustration of our neural network. 21
3.3 Mean Gini Decrease when each variable is left out of the Random Forest Model 23

4.1 A visual of UVA's network topology and our data collection pipeline. . 27
4.2 The components of our traffic processing pipeline. 28
4.3 Visualization of load balancing in Bro. 32
4.4 A configuration file for Bro that load balances traffic across 9 workers . 33
4.5 Packet Drop Rate and Link Utilization After Applying Load Balancing and Log Reduction 35
4.6 Packet Drop Rate and Link Utilization After applying Subnet Filtering 36
4.7 Performance spirals out of control over time 36
4.8 CaptureLoss Before and After Turning Off Offloading and Checksums 38
4.9 Summary of All Optimizations 38

5.1 An overview of the labeling pipeline 42
5.2 Samples of Abuse Ransomware Feed and routeviews file from Apr 10th, 2019 43
5.3 Converting Entities between URLs, Domain Names, IP Addresses, ASNs and Subnets in Python 44
5.4 Example of parsing process for a url from the Abuse-URL feed 44
5.5 Schema for the Labeling Database 45
5.6 Algorithm for propogating new record information into database . . . 45

5.7 Process for labeling a connection with a database containing blacklist, whitelist, and honeypot data 46
5.8 Sample honeypot data in CIF log from Mar 28, 2019 48

List of Tables

3.1 Malware Description 14
3.2 Traffic Type Distribution in Dataset 14
3.3 Samples Used in Dataset 15
3.4 Evaluation Metrics 21
3.5 LOBO-CV F1 score 23
3.6 LOBO-CV F1 Score for Ensemble Model 24

4.1 Minimal Set of Logs for Traffic Capture 34
4.2 Subnets Used in Traffic Filter 35

5.1 Feeds Collected for Labeling Database 50
5.2 Label Mappings 51
5.3 Portion of Connections from the US By Class 53

List of Abbreviations

ACK	Acknowledgement Flag (TCP)
ASN	Autonomous System Number
AUC	Area Under the Curve
C&C	Command and Control
CDN	Content Delivery Network
CV	Cross Validation
FFT	Fast Fourier Transform
FTP	File Transfer Protocol
FPR	False Positive Rate
HAR	High Access Rate
HTTP	Hypertext Transfer Protocol
ICMP	Internet Control Message Protocol
IDS	Intrusion Detection System
LOBO-CV	Leave One Bot Out Cross Validation
LR	Logistic Regression
NB	Naive Bayes
NIC	Network Interface Card
NN	Neural Network
pcap	packet capture
RELU	Rectified Linear Unit
RF	Random Forest
ROC	Receiver Operating Characteristic
SBB	Symbiotic Bid-Based Model
SSL	Secure Socket Layer
SYN	Synchronization Flag (TCP)
SVM	Support Vector Machine
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UVA	University of Virginia
VM	Virtual Machine

Chapter 1

Introduction

1.1 Introduction

Cybercrime has become one of the most pressing issues of our time. Cyber-attacks cost businesses on average $11 million annually and there are no signs of slowing down, as the amount of attacks has doubled in the last 5 years [13]. Moreover, attackers are employing increasingly sophisticated methods, targeting government, business, and academic institutions alike. In particular, UVA has been the target of many notable attacks in the last decade. In 2007, hackers from China infiltrated UVA IT systems and stole over 5400 social security numbers of students and faculty. [14] In 2015, attackers hacked into UVA mail servers and compromised the mail accounts of multiple university employees. Universities across the country are subject to attacks like these on a regular basis, so it's essential to develop techniques to detect and neutralize these threats.

The traditional defense against cybercrime is an Intrusion Detection System (IDS), which examines network traffic for malicious or anomalous behavior. In most cases, these systems use signature-based detection, relying on pre-defined rules and attack signatures to detect intrusions. This strategy is woefully inadequate in an age where cyber-attacks are constantly evolving. A more promising approach is anomaly detection, which models host or endpoint behavior on a network in order to detect malicious traffic. In these systems, predictive models are trained to learn behavioral patterns from the data, as opposed to being told exactly what to look for.

1.2 Objectives

The objective of this thesis is to lay the foundation for an end-to-end, anomaly-based Intrusion Detection System for enterprise networks. This objective is contingent on developing a strong data pipeline that captures, pre-processes and labels network traffic on a continuous basis. We build such a pipeline for UVA network traffic. While constructing this pipeline, we set out to evaluate multiple machine learning approaches for anomaly detection on open-source datasets. In short, our goals are:

- Build an end-to-end Intrusion Detection System for enterprise networks
  - Build a robust pipeline to process and label UVA network traffic
  - Concurrently develop threat detection methodology on open-source data

1.3 Contributions

The contributions of this thesis are as follows: (i) A pre-processing pipeline that continuously collects, anonymizes, and labels network traffic; (ii) A scalable labeling system that can be used for any IP-based labeling task; (iii) a methodology for training and evaluating models for botnet detection. This thesis will hopefully pave the way for real-time anomaly detection for the UVA network as well as other enterprise networks.

1.4 Thesis Organization

The rest of the thesis is organized as follows. Chapter 2 explains important networking concepts relevant for understanding the thesis and surveys related work. Chapter 3 details our methodology for building and evaluating supervised learning models for botnet detection. Chapter 4 explains our UVA network traffic processing pipeline, and details all the optimizations we had to make to achieve successful high-speed packet capture. Chapter 5 describes a fully autonomous IP-based traffic labeling pipeline that we use to label our UVA network traffic for supervised learning. Finally, Chapter 6 concludes the thesis and proposes avenues for future work.

Chapter 2

Background & Related Work

This chapter provides a cursory background of networking concepts necessary to understand this thesis and presents a literature review of related work in the field. The chapter is organized as follows: Section 4.1 explains fundamental networking concepts and terminology; Section 4.2 describes Intrusion Detection Systems; Section 4.3 describes machine learning approaches to anomaly detection; Section 4.4 describes high-speed packet capture solutions used to pre-process network traffic; Section 4.5 describes approaches to labeling network traffic for supervised learning.

2.1 Networking Concepts & Terminology

I refer to many networking concepts throughout this thesis assuming the reader understands them. Thus, I will briefly the most salient networking concepts that are referred to in later sections. If you are already familiar with networks, feel free to skip to section 2.2.

In every network connection, there is a source S (usually a client) and a destination D (often a server). Collectively, they are known as *endpoints*. Endpoints communicate back and forth by sending digital chunks of information to each other. These chunks are known as *packets*, which are the smallest unit of data that can be sent across a network. Each packet contains a header, which specifies S and Ds' IP addresses and a payload containing all the data to be sent to the destination. Each packet is associated with a certain *protocol*, or communication standard. In some protocols such as Transmission Control Protocol (TCP), hosts send *flags* to each other, which act as formal keywords marking different stages of the connection.

During the connection, data is sent between two *ports*: A source port at S and a destination port at D. A port identifies a server or client process to which the TCP/UDP or other protocols send packets. While a server such as a web server can have thousands of ports, some ports are commonly associated with specific services. For example, HTTP uses port 80, HTTPS uses port 443, and SSH uses port 22. As such, server ports often indicate the activity associated with a connection.

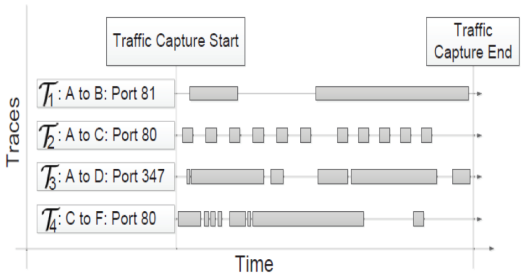

FIGURE 2.1: An illustration of different traces. Graphic courtesy of Botfinder [22]

Packet traffic can be recorded in *pcap* (packet-capture) files and directly analyzed with open-source software. While pcap data provides highly granular information for forensic analysis, it accumulates quickly and requires significant storage space. An alternative to pcap data is *flow* data, which contains aggregate level statistics about the packets exchanged between a source and destination over a certain time interval [22]. Since they are summary statistics, they require significantly less space than pcap data, but they lack the detailed information contained in packet payloads. Each flow is identified by a *five-tuple*, i.e. source IP, destination IP, source port, destination port, and protocol. In client-initiated flows, the client is the source and the server is the destination. Conversely, for flows from server to client, the server is the source and client the destination. For this reason, we often use client/server terminology instead of source/destination. Client ports are usually ephemeral, meaning they change with time. Thus, flows are often identified in practice by a *four-tuple* consisting of its client IP, server IP, server port, and protocol [22].

Finally, flows can be further aggregated by four tuple into *traces*, which represent all traffic between S and D associated with the same server port and protocol. (See Fig. 2.1). Analyzing traffic at the trace level facilitates the computation

of highly useful metrics for anomaly-based detection because we can analyze behavior across connections flows. In our work, we extract time-based, volume-based and flag-based features from traces between endpoints in order to detect malicious traffic.

2.2 Intrusion Detection Systems

Intrusion Detection Systems (IDSes) play a crucial role in keeping enterprise networks safe. Their primary objective is to monitor live network traffic and report any anomalous or suspicious behavior to network administrators. However, some IDSes possess the ability to react to certain threats based on pre-defined rules [5]. Often times, they collect useful traffic logs that can be used by experts after an attack to determine the root cause. Most existing IDSes are *signature-based* i.e. they depend on pre-defined rules to detect cyber threats. However, many attackers have learned to circumvent these rules, and as such, signature-based systems often miss zero day attacks. A more promising approach as of late has been to build *anomaly-based* IDSes, which leverage statistical models and real network data to derive threat detection rules. While they are often more difficult to deploy, anomaly-based systems tend to outperform signature-based IDSes because they learn new patterns from the data that can generalize to new attack types. This thesis presents the foundation for such an IDS that leverages supervised learning models for malicious threat detection.

2.3 Machine Learning Approaches to Threat Detection

As cyber attacks become increasingly more sophisticated, intrusion detection companies are turning more and more to machine learning to build their systems. According to P&S market research, cyber companies will spend more than $6 billion on Artificial Intelligence by 2023. [39]. Academic research in this area has exploded too, focusing mainly on building attack detection models [11]. While there are many different types of cyber attacks, most of them depend on having a strong botnet, or network of infected machines, to succeed. Thus, detecting botnet traffic can help mitigate a variety of security threats.

A considerable number of studies have been conducted to create machine learning models based on a limited number of botnet families. In these studies, a separate model is often created for each family, or one model is created for 2-3 botnet families. For example, Hadaddi et al. [25] developed C4.5 and Naive Bayes models to detect HTTP-based botnets using two families: Zeus and Citadel. Eslahi et al. [16] proposed the use of HAR (High Access Rate) and LAR (Low Access Rate) filters, which are designed to provide upper and lower bounds on suspicious periodicity. The model was based on two botnet families: BlackEnergy and Bobax. Wang et al. [45] created three clustering models for Kraken, BlackEnergy and Zeus. Lu et al. [33] combined network traffic analysis with hidden Markov models to differentiate the behavior of Zeus C&C communication from normal traffic. Torres et al. [44] trained a threat detection model on open-source data from two botnet families: (DonBot and Neris). Finally, the authors of Botfinder [22] proposed a machine learning based system that calculates statistical features similar to the features we use in our Botnet study.

Some botnet detection models require inspecting packet payloads to extract the required information for detection [17, 31, 24, 12]. For example, Etemad et al. [18] proposed a method to separate IRC traffic from HTTP traffic by light payload inspection. Most feature extraction algorithms regard malware detection as a regular machine learning problem and mainly aim at improving the final detection performance. Our models, however, are based on trace level information that does not require payload inspection. Soniya et al. [42] trained a neural network classifier by running 120 botnet malware samples including Pushdo, Banbra, BlackEnergy, Sasfis, Bifrose, Dedler, and Zeus. Their model achieved a false positive rate (FPR) of 2.5%. Bilge et al. [30] proposed a botnet detection system called DISCLOSURE that extracts three types of features from Netflow: size, host access pattern, and temporal behavior. The features are based on their hypothesis that C&C communication happens in predictable patterns that can be identified with behavioral and temporal metrics. However, while their final model achieved a false positive rate less than 1%, its accuracy was less than 65%. Finally, Haddadi et al. [19] analyzed Netflow data from three Botnets – Zeus, Conficker and Torpig. Flow features such as duration,

number of packets, number of bytes, flows and bits per second were used to classify malicious traffic. A C4.5 Decision tree model and a Symbiotic Bid-Based (SBB) model were trained and tested where the false positive rates were on average 5-10%, depending on the botnet family. In Chapter 3, we demonstrate that our best model can achieve a FPR of less than 1% on similar traffic.

2.4 High Speed Packet Capture Solutions

While plenty of studies deal with applying machine learning to network data, few explain how the data was actually collected. In most cases, they either were provided the dataset by an enterprise [36, 30] or used open source datasets [19]. Those who collect their own traffic have historically narrowed the scope of analysis to single-link traffic which can easily be managed with standard packet capturing tools [15]. In contrast, building a threat detection model at the enterprise level requires a custom, air-tight traffic capturing solution across multiple links with minimal packet loss at high speeds.

While there are some notable commercial products for high speed packet capture, they are often expensive and/or difficult to deploy. However, some papers have shown that lossless high-speed packet capture is possible by leveraging open source software. Sandia Labs compare the performance of four low-budget solutions (PF_Ring, PF_Ring Zero Copy (ZC), Sniffer10G, and NetMap) for traffic capture at varying speeds and flow sizes. They found that PF_Ring ZC and sniffer 10G could keep up with traffic speeds up to 10 Gbps without dropping any packets [8]. Gallenmuller et al. evaluate PF_Ring, netmap and DPDK's packet capturing ability under varying CPU loads and buffer size, and found that PF_Ring outperformed the other commercial software alternatives. [23]. Purzynski et al demonstrate it is possible to capture traffic at speeds up to 10 Gbps using Suricata and AF_Packet [37]. Finally, Stofer et al show that network intrusion detection with Bro (now Zeek) is possible at speeds up to 100 Gbps if the traffic is load balanced across a cluster of high performance hosts. [29]. We have also deployed Bro in our pipeline but are currently performing load-balancing on a single host. Despite this limitation, we are still able to process all incoming traffic from our 10 Gig network tap.

2.5 Approaches to Labeling Network Traffic

To build supervised threat detection models, we need to obtain examples of both malicious and benign behavior for training. There are primarily three approaches for obtaining malicious traffic samples. The first is attack simulation. In this approach, a group of attackers attempts to compromise a fake target, and all network traffic is recorded. Subsequently, all traces associated with the attackers are labeled as malicious and are intermixed with benign traffic samples. While attack simulation has been used extensively in the cyber literature [28, 21, 32], it suffers from a few key drawbacks. First, the efficacy of a model built on this data is highly dependent on the quality and fidelity of the simulations. This can be problematic because there is often a difference between the way researchers expect an attack to occur and the way it actual occurs in the wild. The best way to mitigate this bias is to deploy existing malware for the attack simulation. However, even if the simulations are authentic, they are still based on previously well-known and documented attacks, so a model built on this data will likely fail to detect zero-day attacks.

Another frequently used method to record malicious traffic is deploying a honeypot server. Formally, A honeypot is defined as a decoy computer resource whose value lies in being probed, attacked, or compromised [35]. A honeypot emulates a common service or application such as HTTP, SSH, FTP, HTML or a MYSQL database and is usually deployed in an isolated environment independent of the central network. Upon successful login, all the attacker's actions are recorded and written to log files. However, since these machines merely mimic real services, attackers are unable to do any damage to the machine or the network.

Honeypots have been used to generate ground truth in many notable attack detection papers. Antonakis et al used honeypots to capture malware samples in a controlled environment and calculated features from this traffic for their DNS reputation system [6]. Daniel Zammit built a network of honeypots that continuously collected attack data and used it to train a machine learning model for attack detection [47]. Finally, Song et al built an attack detection dataset on over three years of honeypot data and used this data to evaluate an IDS [41]. In our research, we use

honeypots for IP-based labeling. The idea is that if an IP address is observed attacking the honeypot, we can confidently label its traffic as malicious (or at the very least suspicious) if it appears elsewhere on the network around that time. While this implicitly assumes all honeypot traffic is malicious, the assumption is fairly safe because properly configured honeypot servers are run in isolated networks undisclosed to normal network users [35]. While honeypots are advantageous in many ways, one drawback is that sophisticated malware packages are often programmed to detect and ignore them.

The third and simplest way to obtain labeled malicious traffic is by blacklisting. Blacklisting is a technique to protect the network by filtering network traffic by IP address. In production, they are used as a filter to block traffic to and from blacklisted IP addresses. However, in academia, they tend to be used retrospectively to label threats in recorded traffic. These lists are often painstakingly curated and maintained by cybersecurity experts or researchers. As such, the threats on these lists are usually legitimate. However, they suffer from a few notable drawbacks. First, there is little consensus between blacklists. According to a meta-study of over 80 blacklists, the lists had little to no overlap, even though many of them track the same malicious activities [34]. This means the scope of malicious activity, recency and quality of each blacklist differs drastically from list to list. Second, IP addresses are dynamic; they are leased to a host for a fixed amount of time, but after the lease expires, they can be allotted to an entirely different host. This causes most blacklists to suffer from a high false positive rate [43]. Finally, malicious websites are usually dealt with quickly after they've been publicly blacklisted. As such, a blacklist can grow stale over time and is therefore only truly effective if used to label traffic around the time that the incidents were reported. In our system, blacklists are downloaded on a daily basis and are only used to label the concurrent day's network traffic. In the rare case that an IP attacks the network and it is added to a blacklist later that day, the connection would still be flagged as malicious because labeling is performed retrospectively at the end of the day.

Similarly, the most common approach to obtaining benign samples is whitelisting. Related to, albeit distinct from blacklists, whitelists in industry are used to exclude all traffic except that from sources which are known to be secure. In effect,

they act in the opposite way as blacklists; the default behavior is to exclude traffic unless it is to or from a known, safe source. True whitelisting requires manually verifying that a website is legitimate. However, since the majority of active websites are benign, and there are nearly 200 million active websites on the web, this proves to be a challenging task [20]. Instead, most researchers use website visitation as a proxy, assuming that the most popular websites are benign [43]. This assumption turns out to hold for most consistently popular websites. However, malicious sites can sometimes temporarily surge in popularity and as such be falsely whitelisted. One way to mitigate this issue is to filter the whitelists to only include domains that have been on the list for a substantial period of time. We employ this technique with our whitelist, and it has proven to drastically reduce the false positive rate.

Chapter 3

Evaluating Machine Learning Approaches for Botnet Detection

3.1 Introduction

Since we didn't have a traffic processing pipeline last year, we decided to conduct a pilot study on building threat detection models using third party data. We focused specifically on botnet detection, as botnets are leveraged for many different types of cyber attacks. We pulled the data from the website for the Czech Technical University's (CTU) Stratosphere project. Overall, we compiled over 40 samples of malicious and benign traffic. The malicious samples spanned eight different botnet families and were created by researchers downloading and executing real malware in an isolated environment. The whitelist samples were mostly normal traffic recordings of the researchers. The goal of this study was to build classifiers that could distinguish between benign and malicious traffic samples and detect novel botnet attacks using temporal, volume-based, and TCP flag-based features.

We compared the performance of five different algorithms: Random Forest, Logistic Regression, Naive Bayes, SVM and a feed-forward Neural Network. Of these, Random Forest performed far and away the best, achieving both an F1 score and AUC of .99 when evaluated using k-fold cross validation. However, since we were interested in measuring the model's ability to detect zero-day attacks, we designed a custom cross-validation technique called Leave One Bot Out (LOBO) CV, which measures the botnet's ability to detect traffic from a botnet family it hasn't seen before. Although Random Forest performed admirably, it struggled to detect certain

botnet families that other models performed better on. As such, we ensembled the predictions of the two best models (Random Forest and Logistic Regression) in a weighted fashion to produce ensemble predictions. The ensembe model was able to pick up on the strengths of each individual model and in some cases, even improve on the best individual model. Overall, the ensemble model achieved an F1 score of .99 or higher on all but two families (.95 for Zeus and .90 for Bunitu). The results suggest ensembling approach like this could potentially be highly effective at detecting new botnets in the wild.

This chapter is laid out as follows: Section 4.1 provides necessary background and motivates the problem; Section 4.3 states the specific objectives of the project; Section 4.4 describes the data we used in detail; Section 4.5 describes our data processing pipeline; Section 4.6 describes the features we used for our models; Sections 4.7-4.8 describes our models; Section 4.9 contains the results of model evaluation, and finally Section 4.10 concludes the chapter and suggests avenues of future work.

I worked extensively with Rohan Bapat and Abhijith Mandya on this project, and all coding was done in Python and R. Our code, data and results are accessible via Github. [1]

3.2 Background

An important source of cyber-attacks is malware, or malicious software that infects and compromises a host machine. Malware proliferates in different forms, one of them being botnets, which are groups of remotely controlled, compromised machines. These compromised machines, called *bots*, connect to a central server operated by a *botmaster* that gives them instructions to execute. The bots periodically communicate with the botmaster to receive new instructions. This back and forth between bots and botmaster is known as Command and Control Communication*C&C*. A botnet can be leveraged to perform a variety of deadly tasks, such as DDoS (Distributed Denial of Service) attacks, APT (Advanced Persistent Threats) attacks, or phishing attacks[7]. Since they are a necessary tool for many different attacks, stopping them is a top priority from a security standpoint.

[1] https://github.com/am6ku/Anomaly-based-Intrusion-Detection-System/

3.3 Objectives

Our objectives for this research are as follows:

1. Evaluate multiple machine learning approaches for botnet traffic detection
2. Accurately identify botnet traffic while maintaining a low false positive rate
3. Create a detection system that is light-weight, scalable and generalizable to new, unseen attacks

3.4 Data Description

Our data consisted of traffic from eight different bot families. We have used data released by the Malware Capture Facility Project under Stratosphere IPS Project [1]. The researchers used a testbed network topology consisting of a set of virtualized computers to create *malicious* and *normal* network traffic [40]. The repository contains over 150 botnet traffic samples and dozens of normal traffic samples.

Traffic that came to or from any of the known infected IP addresses was labeled as malicious data. The researchers who generated this traffic used open source code specific to eight different families of botnets. In this way, they produced malicious traffic from each of these families with correct, ground truth labels for each bot. Traffic from the known and controlled computers in the network, such as routers, proxies, or switches, was labelled as normal data.

The traffic was captured in different formats including pcap (packet capture), Netflow and Bro logs. For our analysis, we used conn logs, which contain flow-level data and are generated by the network monitoring framework Bro. Overall, our dataset contained over 22,000 traces and around 1.6 million flows.

The specific samples we used are included in Table 3.3. In total, we pulled 36 samples with malicious traffic from eight botnet families and used multiple normal samples. The indices shown in the table are abbreviated versions of the directory names on the website. Malicious traffic directories follow the naming convention *CTU-Malware-Capture-Botnet_**f_index***, while Normal directories are named *CTU-Normal_**f_index***.

Table 3.1 details the different botnet families present in our dataset, while Table 3.2 details the distribution of traces of these botnet families in our dataset.

TABLE 3.1: Malware Description

Family	**Malware Description**	**Year**	**Botnet Size**
Zeus	First widespread banking trojan - used to steal banking information.	2007	3.6M
Conficker	Worm used to infect Windows systems - Has infected computers in over 190 countries	2008	200k
Dridex	Another banking trojan that targeted large American corporations from 2012-2015	2011	100k-500k
Necurs	Massive botnet used to launch ransomware attacks and banking trojans.	2012	6M
Miuref	Trojan that facilitates click fraud and downloads malicious content.	2014	100k
Bunitu	VPN scam that uses infected hosts as a proxy for remote clients.	2014	100k-500k
Upatre	Malware downloader that downloads and executes Dyre Banking trojan	2014	100k-500k
Trickbot	Trojan that leverages HTML and Javascript injections to steal banking information	2016	100k

TABLE 3.2: Traffic Type Distribution in Dataset

Traffic description	**Number of traces**	**% of overall**
Normal traffic	13,514	48%
Bunitu	6,761	24%
Necurs	3,975	14%
Miuref	1,532	6%
Zeus	711	3%
Other malware	1,379	5%

3.5 Data Processing

Figure 3.1 shows our end-to-end pipeline from data collection to modeling. Data processing consisted of four steps. First, each malicious and benign traffic sample

TABLE 3.3: Samples Used in Dataset

File Index	Type	File Index	Type
140-1	Bunitu	238-1	Trickbot
140-2	Bunitu	239-1	Trickbot
141-1	Bunitu	240-1	Trickbot
141-2	Bunitu	241-1	Trickbot
153-1	Dridex	242-1	Trickbot
227-1	Dridex	243-1	Trickbot
228-1	Dridex	244-1	Trickbot
246-1	Dridex	247-1	Trickbot
248-1	Dridex	176-1	Necurs
249-1	Dridex	7	Normal
128-1	Miuref	12	Normal
128-2	Miuref	2	Normal
169-1	Miuref	21	Normal
169-2	Miuref	22	Normal
169-3	Miuref	23	Normal
143-1	Upatre	78-1	Zeus
162-1	Upatre	78-2	Zeus
162-2	Upatre	91	Conficker

obtained form CTU's repository was parsed, filtered, and cleaned so the connections were grouped into traces. Next, feature extraction was performed on each trace to obtain statistics describing flow-level and trace-level behavior. At this point, each feature vector was assigned the label of its parent sample (i.e. benign or malicious). Third, all feature vectors and labels were collapsed across samples to form our modeling dataset. Finally, this data was partitioned into training and test sets, which were used to train and evaluate a host of supervised learning algorithms. The following sections describe the feature engineering process and the modeling techniques we used in detail.

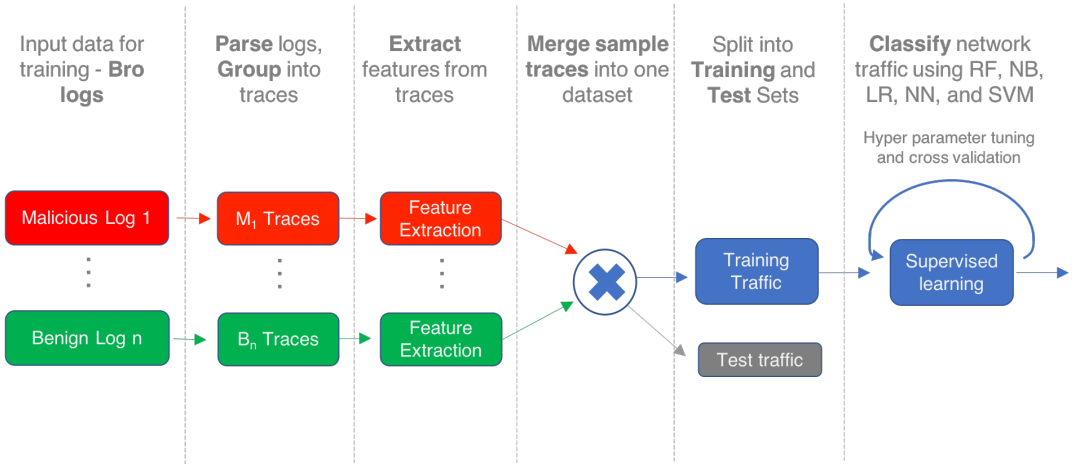

FIGURE 3.1: An illustration of our neural network.

3.6 Feature Engineering

As stated above, the connections within each traffic sample were grouped into traces containing all flows associated with the same four-tuple, or same (source IP, destination IP, destination port, and protocol). We excluded source ports when defining traces because they are ephemeral, meaning they can change throughout the course of a trace. Each trace was collapsed down to a single record containing statistics characterizing the typical behavior between each source and destination on the network. The features we calculated can be generalized to three groups. The first set of features relates to the *volume* of the communication, or the average number of packets and amount of data sent in a typical flow. The second set of features pertains to the *timing* of communication, or the average flow duration and average time interval between successive flows. The final set of features pertains to the state history of connections in the form of flag counts. These counts were obtained by parsing the *state_history* field in the conn logs, which contains a string of successive flags sent throughout the connection. Capital flags were sent from the source, while lowercase flags were sent from the destination. We kept separate counts of source flags and destination flags with the hops of discovering some asymmetries in malicious communication. The features we used were as follows:

1. Average time interval between two consecutive flows in a trace

2. Average Number of source packets and destination packets

3. Average flow duration within a trace

4. Total flag counts (a/c/d/f/h/r/s/t) - Records the state history of connections as a string of letters. The definitions of these flags are as follows –

   (a) a - pure ACK, or Acknowledgement flag

   (b) c - packet with invalid checksum

   (c) d - packet with payload data

   (d) f - packet with the FIN flag set, terminating the connection

   (e) h - SYN+ACK ("handshake") which is standard protocol for establishing a TCP connection

   (f) r - packet with RST bit set which resets the connection

   (g) s - SYN without the ACK bit set (or an open-ended handshake)

We used the mean and standard deviation of the first three metrics - Time interval between start times, number of source and destination packet and flow duration - as features for the training and testing datasets.

3.7 Modeling Overview

We used the training dataset to create five binary classifiers that predict whether a given set of traffic was benign or malicious. We first attempted to create a baseline for classification performance by using a light-weight logistic regression model. We then added Naive Bayes, Support Vector Machine, Random Forest and Neural Network to our umbrella of supervised learning algorithms. In the next section, we explore our modelling process in detail and explain our model-of-choice. We further tested the generalization capability of these models across unseen botnet families in a process which we have termed as Leave-One-Bot-Out cross validation (LOBO-CV). In this scheme, models are trained on all but one family, and the unseen bot is used as a test set. The same performance metrics are used as measurement. Finally, using the best performing models, we devised an ensemble to create a unified model that achieves a strong performance across all families.

3.8 Modeling Techniques

We used five different supervised learning techniques to classify network data and measured their performance. These classifiers were chosen since they represent the most widely used supervised learning models today. Performance of each one varies widely and depends upon the complexity of the given dataset. Since this is a classification problem, prediction accuracy (percentage of correct predictions divided by the total number of predictions), as well as False Positive Rate, were used for evaluation. This evaluation was carried out in two stages. First, the dataset was split into a training set which contained two-thirds of the traces, while the rest was used as the test set. The supervised learning models were trained on the training data and then cross-validated for model selection and model performance assessment. Next, the models were tested using the test dataset. The predictions on test data were used to arrive at the performance metrics for model comparison. All models were trained and cross validated using the Caret package in R[26].

3.8.1 Logistic Regression

Logistic regression (LR) is a widely used classification technique in which the probability of the outcome (trace maliciousness) is related to a series of potential predictor variables by an equation of the form

$$log[\frac{p}{1-p}] = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + ... + \beta_i X_i$$

where p is the probability of the maliciousness, β_0 is an intercept term, β_1.....β_i are coefficients associated with each variable X_1.....X_i. This model assumes that all predictors are related in a linear manner to the log odds of the outcome, or in our case, maliciousness of the trace. To achieve these linear relationships, we performed log transformations on the predictors before modeling. This learning technique has been the method of choice till very recently and still affirms a baseline for statistical modelling. Although it is easy to implement, easy to interpret, and has low computational requirements, it is increasingly seen to be inferior to more recent and complex machine learning algorithms.

3.8.2 Naive Bayes

Naive Bayes (NB) is a widely used framework in statistical modeling using Bayes Theorem. Naive Bayes is the simplest form of a Bayesian network, in which all attributes are independent given the value of the class variable.

$$f_{nb}(E) = \frac{p(C = +)}{p(C = -)} \prod_{i=1}^{n} \frac{p(x_i \mid C = +)}{p(x_i \mid C = -)}$$

where C is the label of maliciousness and x are the predictor variables. This is called conditional independence. However, this assumption is rarely true in most real-world applications. Naive Bayes owes its good performance to the zero-one loss function. This function defines the error as the number of incorrect classifications. Unlike other loss functions, such as the squared error, the zero-one loss function does not penalize inaccurate probability estimation if the maximum probability is assigned to the correct class. This means that naive Bayes may change the posterior probabilities of each class, but the class with the maximum posterior probability is often unchanged. Thus, the classification is still correct, although the probability estimation is poor.

3.8.3 Support Vector Machines (Radial)

Support Vector Machines (SVM) separate a given set of binary labeled training data with a hyper-plane that is maximally distant from these binary classes (known as 'the maximal margin hyper-plane'). For our case, where a non-linear separation is possible, they work in combination with a radial kernel, that automatically realizes a non-linear mapping to a feature space. The hyper-plane found by the SVM in feature space corresponds to a non-linear decision boundary in the input space.

$$\gamma = min_i y_i \Big\{ \langle w, \phi(x^i) \rangle - b \Big\}$$

where the margin γ is maximized, the hyperplane (w, b) with input data x_i with the corresponding label y_i. The quantity $(w, \phi(x^i)\rangle - b)$ corresponds to the distance between x_i and the decision boundary.

3.8.4 Random Forest

Random Forest (RF) is an ensemble of decision trees where the features of each tree are chosen by a random vector sampled independently and with the same distribution for all trees in the forest. The error rate for forests converges to a limit as the trees grow asymptotically. Every classification tree in the forest casts a vote for the sample after which the majority vote determines the class of the sample. The strength of the individual trees in the forest and the correlation between them determine error for random forest classifiers [9]. Internal estimates monitor error, strength, and correlation, which are used to measure variable importance. It also holds better sway in terms of identifying underlying variable interactions.

3.8.5 Fully-Connected Neural Network

In a fully-connected, feed-forward neural network (NN), the input data is sent through a series of fully-connected layers of neurons that ultimately produce a continuous output between zero and one. Each connection has an associated weight that represents the strength of the connection. The higher the weight, the stronger the connection. In the brain, a neuron "fires" if the combined input received from surrounding neurons exceeds an activation threshold. Similarly, in a NN, a neuron in one layer will send a strong signal if the weighted sum of the input it receives from the previous layer exceeds a certain threshold. The strength of this signal is determined by an activation function which can take many forms. Original neural networks used Haveside step functions, but most networks nowadays use Sigmoid (logistic) or Rectified Linear Unit (RELU) which is a piecewise-linear activation function. We used Sigmoid activation functions for all neurons in our network. Our network architecture is shown in Figure 3.2.

As shown in Figure 3.2, there are three types of layers: an input layer, two hidden layers, and an output layer. The two hidden layers have 10 and 5 neurons respectively. The input layer feeds the raw data to the nodes in the first hidden layer. Each node $H_{1,j}$ computes a weighted sum of its received inputs:

$$H_{1,j} = \sum_{i=1}^{n} w_{1i} x_i + b$$

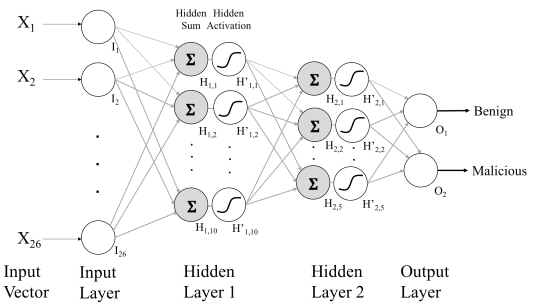

FIGURE 3.2: An illustration of our neural network.

where b is a bias term. The result is passed to a sigmoid function that converts it to a 0-1 scale:

$$H'_{1,j} = \frac{1}{1 + e^{-H_{1,j}}}$$

This process is repeated for the second hidden layer. Finally, a weighted sum of the second hidden layer's output is computed which assigns the malicious and benign probabilities. We trained the neural network with standard back-propagation and Stochastic Gradient Descent.

3.9 Evaluation and Results

3.9.1 K-fold CV

TABLE 3.4: Evaluation Metrics

Model	L.R.	N.B.	SVM	R.F.	N.N.
F1 Score	0.96	0.73	0.86	0.99	0.77
AUC	0.90	0.60	0.99	0.99	0.89

We first evaluated our models with a traditional k-fold cross validation scheme. As shown in Table 3.4, Random Forest was the best performing model, proving to be competitive with results obtained by other detection systems such as Disclosure or Botfinder. It achieved the highest F1 Score across every fold and dominated all other models at every classification threshold, as shown in the ROC plot. SVM was comparable to Random Forest in terms of AUC but had an F1 score of 0.86. The next tier of models included Logistic Regression and the neural network, which achieved

scores of 0.96 and 0.77 respectively. We believe the neural net's performance would have improved if we had more data and trained it for more epochs. However, in both cases, the models missed a significant portion of malicious traffic, rendering them ineffective for real world anomaly detection.

While Random Forest was clearly the best performing model, it remains to be seen which variables are driving this success. Figure 3.3 shows a variable importance plot generated by the Random Forest model. This plot measures the mean accuracy decrease when each feature is left out of the random forest model. The plot suggests that *mean_src_pkts* is the most important feature for Random Forest in discriminating between malicious and benign traffic. Additionally, certain metrics such as *mean_intvl* and *stdev_intvl* bear substantial importance in the Random Forest's decision making process.

3.9.2 LOBO CV

While the previous results are promising, they assume that we have access to samples from every botnet family a priori. However, malware is constantly changing, so our system will likely encounter novel botnet variants absent from the training data. Thus, it's imperative to understand how well our models classify traffic from an unseen bot. To measure this, we implemented a validation scheme called Leave-One-Bot-Out Cross-Validation (LOBO-CV). In contrast to traditional cross validation where the models have seen traffic from all botnets, each model in the LOBO-CV is trained on seven bot families, and the eighth unseen bot is used as a test set. Unsurprisingly, all our models suffered performance-wise. Naive Bayes was the worst performing model while the Neural Network and Radial Kernel SVM performed comparably. SVM's score is notably lower than traditional cross-validation, suggesting it over-fit the data. Finally, Logistic Regression and Random Forest were the top performers. While the Random Forest scored greater than 0.99 on all but one bot family - Zeus. In contrast, Logistic Regression had its best F1 score of 0.95 on Zeus. This suggests that the algorithms were picking up on inherently different patterns in the data. This led us to believe that creating an ensemble of Random Forest and Logistic regression could improve overall performance.

TABLE 3.5: LOBO-CV F1 score

Family	LR	NB	SVM	RF	NN	Family Average
Miuref	0.84	0.51	0.71	**0.99**	0.79	.82
Bunitu	**0.81**	0.47	0.72	**0.81**	0.71	.74
Upatre	0.94	0.57	0.90	**1.00**	0.95	.92
Dridex	0.84	0.53	0.85	**1.00**	0.81	.87
Necurs	0.82	0.48	0.91	**0.99**	0.82	.88
Trickbot	0.82	0.55	0.87	**1.00**	0.83	.88
Conficker	0.81	0.48	0.83	**1.00**	0.78	.85
Zeus	**0.95**	0.59	0.77	0.68	0.80	.74

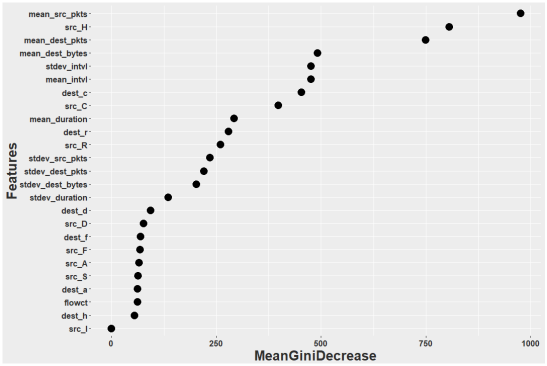

FIGURE 3.3: Mean Gini Decrease when each variable is left out of the Random Forest Model

3.9.3 Ensemble Results

Based on the Leave-One-Bot-Out results, we decided to use ensemble modeling to improve the Random Forest's overall performance. To do this, we combined Logistic Regression and Random Forest predictions in a weighted fashion to produce ensemble probabilities. The predictions took the form

$$y_{ENS} = \alpha * y_{RF} + (1 - \alpha) y_{LR}$$

where α is a tunable weight parameter between 0 and 1, and y_{RF} and y_{LR} are random forest and logistic regression predictions respectively. These probabilities were converted to class labels (malicious or benign) via a cut-off threshold. Both α and the cut-off threshold were learned by using grid search to find the combination that

yielded the highest F1-Score. Table 3.6 compares the ensemble's leave-one-out performance to that of Random Forest and Logistic Regression.

TABLE 3.6: LOBO-CV F1 Score for Ensemble Model

Family	Ensemble	RF	LR
Miuref	0.99	0.99	0.84
Bunitu	0.90	0.81	0.81
Upatre	0.99	0.99	0.94
Dridex	1.00	1.00	0.84
Necurs	0.99	0.99	0.82
Trickbot	0.99	0.99	0.82
Conficker	1.00	0.99	0.81
Zeus	0.95	0.67	0.95

As seen in Table 3.6, Bunitu's F1 score rose from 0.81 to 0.89, while all other scores were the same. It seems the ensemble was able to use the best aspects of both models to increase or maintain detection performance across all unseen botnet families. Since Bunitu formed the largest part of our data, we feel this ensemble has the potential to improve our predictive power when we encounter a larger, more variable dataset with unknown bot families.

3.10 Conclusions and Applications to UVA Data

We achieved two main objectives through our work - first, we identified useful features to classify malicious traffic, and second, we compared the performance of five different supervised learning models. We found that *mean_src_pkts*, *mean_interval*, and *std_interval* play the most significant role in discerning malicious traffic.

Our results showed that Random Forest was the superior model with an F1 Score of **0.99**. Random Forest also proved to be more robust than the other models, as the LOBO-CV results demonstrated it could generalize to most unseen bots. This is an important finding because it suggests our model is robust enough to detect zero-day attacks, which often challenging to detect. Our Logistic Regression model performed comparably to, and in some cases better than, Random Forest, suggesting it could be a strong light-weight substitute in real-world applications. The results

from the Neural Network were slightly underwhelming but we hypothesize that a larger training set, better topology, and increased training epochs would greatly improve the model. We also ensembled the Random Forest and Logistic Regression models to produce overall performance that beat any singular model for all families.

While our results are encouraging, they might not generalize well on real network traffic, as our datasets had a balanced distribution of malicious and benign traffic. In the wild, malicious traffic usually constitutes a small fraction of the entire dataset. Class imbalance such as this poses a significant problem for standard machine learning algorithms. When the class distribution is heavily skewed, the model often fails to detect any positive instances because it can achieve high accuracy by predicting the majority class every time.

While there are many ways to combat this problem, the most common solutions fall into three categories. The first involves balancing the class distribution by either oversampling the minority class, under-sampling the majority class, or both. However, this fundamentally alters the class distribution of the training data to facilitate learning while leaving the test set distribution unchanged. This phenomenon is called "Dataset Shift" and often leads to a drop in test performance [27]. An alternative approach is to do cost-sensitive learning, where each error type has a different cost and the goal is to minimize a weighted sum of the error costs. This allows one to train on a realistically balanced dataset while improving the true positive rate. However, this approach only works when costs of each error type can be quantified and when the costs are significantly different. The third solution is to add more features. If the right features are added, a class boundary could emerge where it previously didn't exist which could drastically improve performance. For botnet detection, there is evidence that temporal metrics like host access patterns and Fast Fourier Transforms (FFT) can further improve our attack detection [22]. While time-consuming, this is a safe approach with little down side.

In future work, we plan to extend this approach to identify botnet traffic across a large network like the University of Virginia. Additionally, we plan to tackle the imbalance problem by adding more features, experimenting with different sampling rates, and potentially adding error costs if we can find a way to quantify the cost of type 1 and type 2 errors for botnet detection.

Chapter 4

UVA Traffic Processing Pipeline

In contrast to many network-based tasks, anomaly detection only truly works if the observer has access to all the data. This is because malicious traffic constitutes such a small minority of overall traffic that even a small loss can possibly result in threats slipping through the network undetected. Therefore, to perform anomaly detection on enterprise-level network traffic, one needs a traffic collection pipeline that drops little to no traffic. In an ideal world, this capturing pipeline would be lossless (i.e. zero packet drops). In practice, however, truly lossless packet capture at 10 Gbps is practically infeasible. Thus, the pipeline should yield accurate data while dropping as close to zero packets as possible. The next two sections present the foundations for such a pipeline. The system is end-to-end in the sense that it captures, pre-processes, then labels network traffic for supervised learning, all while achieving minimal packet loss. The pipeline is broken down into two steps: traffic processing and traffic labeling. The end result is a system that produces labeled network data for supervised learning. This chapter explains packet capturing and pre-processing, while Chapter 5 explains traffic labeling.

This chapter is organized as follows. Section 4.1 gives a high level overview of the end-to-end traffic collection system. Section 4.2 describes packet forwarding with the Gigamon, the first step of the pipeline. Section 4.3 describes packet processing, the next and most intensive step of the pipeline. This section contains four parts; Sections 4.3.1 describes each component of the packet processing system, and introduces Bro, the IDS we use to process packet data; Section 4.3.2 considers the challenges of high-speed packet capture; Section 4.3.3 defines the loss and utilization metrics used to evaluate system performance; finally, Section 4.3.4 explains the

optimizations we made to overcome the challenges of high-speed packet capture and contains our most significant contributions to the field. Section 4.4 describes traffic anonymization, the final step of the processing pipeline. Lastly, Section 4.5 concludes the chapter and sets the stage for Chapter 5 (traffic labeling).

Code for most of these optimizations including shell scripts, Bro scripts, and config files can be found on the UVA High Speeds Network repo [1].

4.1 System Overview

Figure 4.1 contains an overview of our network configuration and data collection pipeline. First, UVA traffic is mirrored from two edge routers to the Gigamon. This device then combines and forwards all traffic to a machine called Ivy bulwark. At Ivy Bulwark, packets are fed into a network monitoring platform called Bro (now known as Zeek), which parses the raw packet data to produce a variety of traffic logs. These logs are transferred to a separate machine on the Ivy network where they are anomymized with a program called CryptoPan. Finally, the anonymized logs are sent to the Rivanna High Performance Cluster (HPC) where they are labeled via blacklist, whitelist, and honeypot feeds for supervised learning. What follows is a detailed explanation of each component of the traffic collection system and the challenges we had to overcome to successfully perform high-speed packet capture.

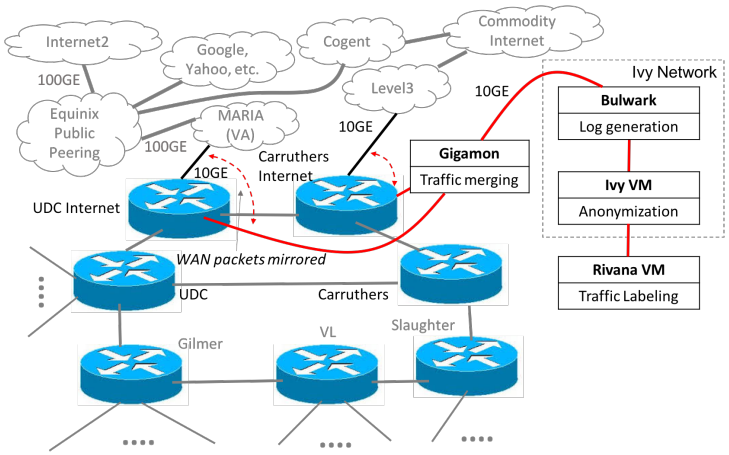

FIGURE 4.1: A visual of UVA's network topology and our data collection pipeline.

[1] https://github.com/UVA-High-Speed-Networks/cyberDevelCode

4.2 Packet Forwarding with the Gigamon

As shown in Figure 4.1, UVA is connected to the internet via two edge routers: one at the University Data Center (UDC) and another at Carruthers. The UDC Internet router connects to MARIA, while the Carruthers Internet router connects to Level 3 applications. Together, these routers handle all the inbound and outbound traffic on the UVA network. This bidirectional traffic is mirrored from both routers via 10 GE links to the Gigamon, a traffic-forwarding device. All UDC packets are mirrored to one port on the Gigamon, while all Carruthers packets are mirrored to another. The Gigamon then merges, filters (if nececssary), and forwards this traffic to Ivy Bulwark via another 10 GE link.

4.3 Packet Processing

The next step of the pipeline involves processing packet data to obtain traffic logs. This process consists of three phases, which are shown in Figure 4.2 and described in detail below.

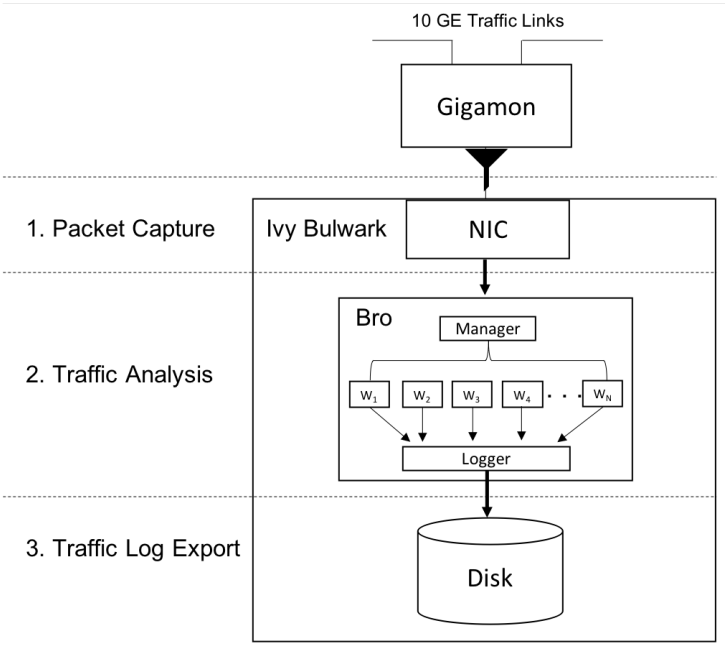

FIGURE 4.2: The components of our traffic processing pipeline.

4.3.1 Processing Steps

1. **Packet Capture**

   As stated above, the Gigamon sends traffic to Ivy Bulwark via a 10 GE link. This link connects to a 10 Gbps interface called `em1`, which is bound to an Intel X520 Network Interface Card (NIC). The NIC captures raw packet data from the link and sends it up to the kernel where it is ultimately redistributed to other applications.

2. **Traffic Analysis**

   Next, the raw traffic is sent to Bro which analyzes incoming traffic for patters and outputs a variety of useful traffic logs characterizing network behavior. [2] (**NOTE**: As of Oct. 2018, Bro became known as Zeek. While the names are interchangeable, the platform will be referred to exclusively as Bro throughout this thesis to reduce confusion). Traffic is first received by the Bro manager, then redistributed to Bro workers. The actual analysis is carried out by Bro workers, which extract log-level data from incoming packets and export them to a central logger as shown in Figure 4.2.

3. **Log Exporting**

   Finally, the logger combines log data across workers and writes it to disk on an hourly basis. The logger can export over 60 unique logs that can be used to characterize network traffic and detect threats on the network. Most, but not all, logs map one-to-one to specific protocols (e.g HTTP logs, SMTP logs, etc). One important exception is a Connection log (Conn log), which contains data about all TCP, UDP and ICMP connections observed on the network. While we collect multiple logs, the processing pipeline was built primarily to handle connection logs. However, future iterations will expand include others such as HTTP and SSL logs.

4.3.2 Processing Challenges

Capturing and processing network traffic at speeds up to 10 Gbps with one host is quite challenging. Even when all traffic arrives successfully to the machine, there

are still many bottlenecks where problems can arise. The first of these is the network interface(s). Typical network drivers are unable to keep up with high speeds, so successful high-speed packet capture requires the use of special, properly configured Network Interface Cards (NICs). Second, most traffic collection tools were built to handle loads up to 1 Gbps and thus cannot keep up with higher speeds. This is true for Bro, which is configured by default to handle all traffic serially on a single core. Finally, the log creation process is highly taxing with respect to compute and memory resources. These bottlenecks required us to perform significant fine-tuning and optimizations to enable near lossless packet capture.

4.3.3 Metrics

We performed our optimizations with respect to two metrics: packet drop rate and CaptureLoss. We also measured link utilization to understand the relationship between traffic load and the loss metrics. Each metric is explained in detail below.

Packet Loss (Packet Drop Rate)

The formula for packet loss is simply

$$\frac{pkts\_dropped}{pkts\_dropped + pkts\_received}$$

It is a local measure that can be calculated at any point in the pipeline. Different tools can be used to obtain this measurement such as `if_config` at the NIC, or a bro command called `netstats`, which queries each worker for packet drops. However, since it is a local measure, it fails to capture losses downstream after the measurement is taken.

CaptureLoss

Bro defines its own loss measure called `CaptureLoss`. In contrast to packet loss, CaptureLoss is an end-to-end loss metric which measures the number of sequence gaps in TCP traffic over the entire connection period[3]. Each time Bro receives a new TCP packet for a certain connection, it is recorded as an event. The event is considered a gap if the packet's ACK number is higher than expected. This indicates that

intermediate packets with lower ACK numbers were dropped. Thus, they define CaptureLoss as the percentage of events which are "gap" events, or

$$100 * \frac{num\_gap\_events}{total\_events}$$

While it is reportedly an accurate measure of overall loss [10], it reveals nothing about what caused the loss. While it is fundamentally different metric than packet loss, both indicate that traffic is being dropped at some point in the pipeline. Thus, we made our optimizations with the intent to minimize both metrics as much as possible.

Link Utilization

Another important metric that is often correlated with packet loss is link utilization. Link utilization is the amount of traffic passing through an interface per second. As one would expect, packet loss is more likely to occur at high link utilization rates. We calculate it by sampling the output of cat `/proc/net/dev` at two different points in time and computing the difference. This command returns how many bytes an interface has seen up to that point in time. So if we get the byte count to be B_t at time t, and $B_{t+\epsilon}$ at some later time $t + \epsilon$, then the utilization would be the change in bytes over the change in time, or

$$\frac{B_{t+\epsilon} - B_t}{\epsilon}$$

4.3.4 Optimizations

Without any optimizations, our packet loss was anywhere from 70-90%, depending on the traffic load. Similarly, our initial CaptureLoss readings were alarmingly high, reaching as high as 95% during peak hours. However, each optimization incrementally improved performance until we were able to reduce both packet drops and CaptureLoss down to nearly zero.

Below are the key improvements we made to the traffic capturing system to reduce both forms of losses. They are presented in chronological order, as each optimization was applied sequentially. While there were no formal experiments conducted to quantify the improvement of each optimization, many ad-hoc plots

were made to qualitatively assess the performance impact and are shown in Figures 4.5,4.6,4.7 and 4.8. To aid the reader, the name of each optimization is followed by the component of the pipeline which it affected. Finally, the optimizations are summarized by group in Figure 4.9.

Increasing the Buffer Size (NIC)

The first place we were losing packets was at the NIC. This loss was detected using `ifconfig`. During heavy traffic loads, the NIC buffer got overwhelmed because it was too small, which caused packets to drop from the queue. However, this was easily solved by increasing the NIC's buffer size to 4096 bytes. Applying this change reduced drops reported by `ifconfig` to zero. However, it had zero impact on CaptureLoss.

Load Balancing (Bro)

By default, Bro is configured to process all the traffic serially with one worker (i.e. via one CPU). However, to capture traffic at speeds above a few hundred Mbps, it must be configured to perform load balancing. Otherwise, Bro will get overwhelmed and drop most of the traffic (in our case, around 80%). The idea is to divide a heavy load as evenly as possible across multiple workers in parallel so that no single CPU gets overwhelmed.

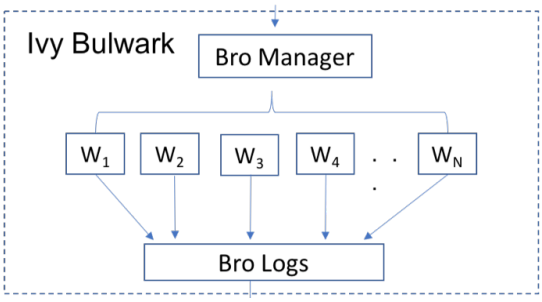

FIGURE 4.3: Visualization of load balancing in Bro.

Figure 4.3 demonstrates how load balancing works in Bro. The interface(s) forward traffic through the kernel to Bro manager. The manager then load balances the traffic across n different worker threads, partitioning the traffic using a flow-based hash function which sends all traffic from the same flow to the same worker.

Bro supports many methods for load balancing, such as kernel modules `PF_Ring`, `DNA/Libzero` (now called Zero Copy), and `AF_Packet`, a low-level module that creates packet sockets to read traffic off the NIC. While we experimented with different solutions, we ultimately settled on `AF_Packet`, as it achieved the best performance.

The manager, logger, and worker threads are defined in a config file (called `node.cfg` by default) like the one shown in Figure 4.4. In this case, each worker is pinned to a unique CPU. We had to choose the CPUs carefully to avoid hyper-threading (i.e. they are all on the same NUMA node). In the example below, traffic is captured off of `em2` and is partitioned across nine workers, each running on their own CPU. `AF_Packet` is activated by prefixing the interface with the chosen load balancer. The load balancing method is set to custom for `AF_Packet` but is set to `pf_ring` if using `pf_ring`. `AF_Packet::FANOUT_HASH` tells Bro to split traffic by their IP tuple (i.e. flow-based hash). Load balancing greatly reduced packet loss but had no impact on CaptureLoss. Instructions for load balancing were pulled from J Gras's Github repo [2].

```
[manager]
type=manager
host=localhost

[proxy-1]
type=proxy
host=localhost

[logger]
type=logger
host=localhost

[worker-1]
type=worker
host=localhost
interface=af_packet::em2
lb_method=custom
lb_procs=9
pin_cpus=1,2,3,4,5,6,7,8, 9
# Optional parameters for per node configuration:
af_packet_fanout_id=23
af_packet_fanout_mode=AF_Packet::FANOUT_HASH
af_packet_buffer_size=128*1024*1024
```

FIGURE 4.4: A configuration file for Bro that load balances traffic across 9 workers

[2] https://github.com/J-Gras/bro-af_packet-plugin

Minimal Log Writing (Bro)

Bro writes up to sixty different logs to a daily directory on an hourly basis. As stated above, this process can be highly memory intensive resulting in loss if the logger fails to keep up with the data it receives from the workers. To mitigate this problem, we turned off all unnecessary log types so the logger could focus on writing out solely the most important types for our research. This was achieved by writing a bro script to turn off all unnecessary log streams and analyzers and invoking this script in Bro's `init` routine. Specifically, we turned off all logs except those shown in Table 4.1. The first four logs collect traffic by protocol, while `stats` collects performance statistics for each bro worker, and `captureLoss` reports the aggregate CaptureLoss across workers at 15 minute intervals. Credit for this optimization goes to Sourav Maji, who initially thought of the idea and wrote the initial Bro script that we deployed on Bulwark.

TABLE 4.1: Minimal Set of Logs for Traffic Capture

HTTP	Conn	SSL	DNS	SSH	Stats	captureLoss

Applying this optimization in tandem with load balancing and increasing the NIC buffer size led to immediate performance improvements. Figure 4.5 shows that after these changes, Bro could handle up to 2 Gbps without dropping any packets for a short period of time. However, Bro's performance began to deteriorate after about 7-8 hours when utilization increased above 2 Gbps. Credit for this idea goes to Sourav Maji.

Subnet Filtering (Gigamon)

To further reduce the strain on Bro, we excluded a substantial portion of incoming traffic by applying subnet filters to the Gigamon. Specifically, filters were writtten to instruct the Gigamon to only forward traffic if it originated from one of the subnetworks shown in Table 4.2. These filters were written and applied to the Gigamon by Jeff Colyer.

As shown in Figure 4.6, applying the filter led to zero packet loss for over a day and a half, a substantial improvement over prior performance. However, it

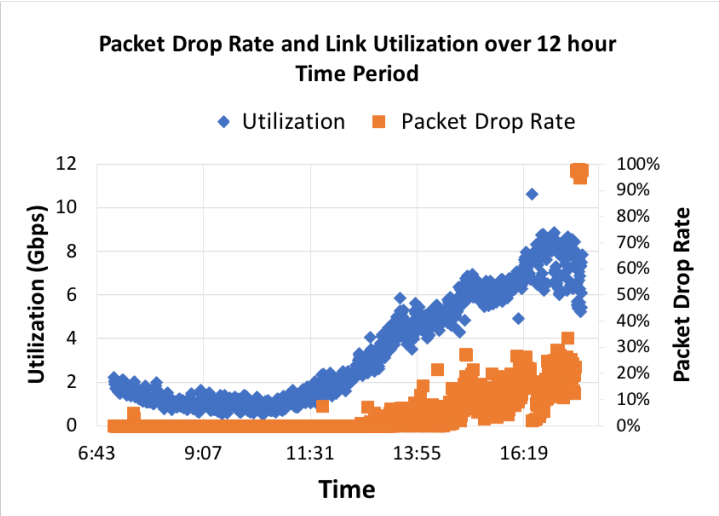

FIGURE 4.5: Packet Drop Rate and Link Utilization After Applying Load Balancing and Log Reduction

TABLE 4.2: Subnets Used in Traffic Filter

Subnet (CIDR Notation)
128.143.0.0/16
137.54.0.0/18
137.54.64.0/18
137.54.128.0/17

seems that something internally with Bro went awry between the 22nd and 23rd. During this time, Bro's packet losses soared to nearly 30% during peak utilization and eventually regressed to zero after utilization subsided. The fact that Bro was able to handle a similar traffic load a day before without loss suggested there was a time-dependent influence at play.

Avoiding Memory Leaks with Daily Restarts (Bro)

Further tests confirmed that Bro's performance would deteriorate after 1-3 days. Initially, Bro would perform flawlessly, but overtime the packet drop rate would either temporarily spike as in Figure 4.6, or completely spiral out of control as shown in Figure 4.7:

In the case of Figure 4.7, the packet drop rate seems to follow a wave-like pattern, ranging from 0 to a certain peak. However, as time progresses, the peak amplitude increases as well, suggesting the problem worsens with time. After much research,

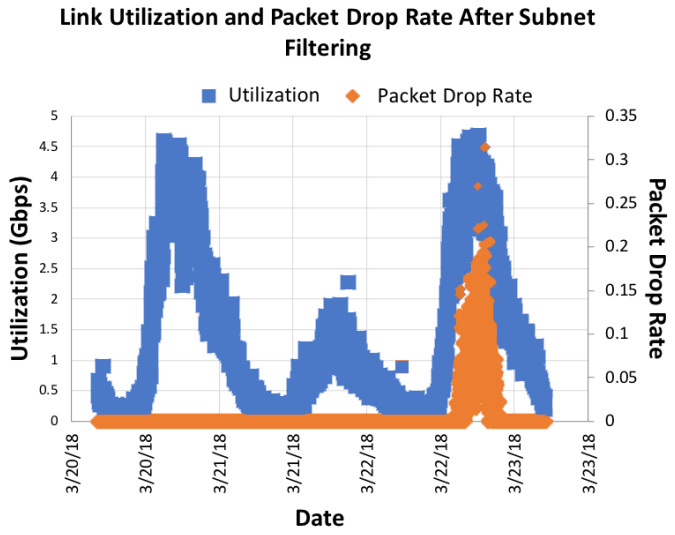

FIGURE 4.6: Packet Drop Rate and Link Utilization After applying Subnet Filtering

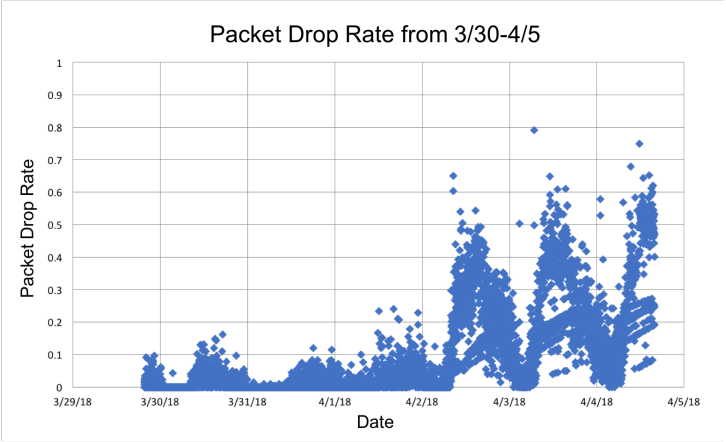

FIGURE 4.7: Performance spirals out of control over time

we concluded it was likely caused by a memory leak in Bro's codebase which gradually consumed more memory over time, leaving less resources for Bro to process incoming traffic. While we unable to pinpoint the root cause, we found that restarting Bro on a daily basis resolved the issue. One might argue this causes gaps in our traffic when Bro is rebooting. However, the losses are minimal, as the rebooting process takes less than a 30 seconds. After applying the daily restarts, we were able to achieve minimal packet loss for weeks at a time.

Disabling RX/TX Offloading (NIC)

At this point, by applying the above optimizations, we had successfully reduced packet loss at all points in the pipeline to acceptable levels. However despite this, our CaptureLoss was still alarmingly high. Together, these facts suggested that all traffic was successfully arriving to Ivy, but Bro was still observing large gaps in the traffic.

We eventually traced the problem to two issues. The first issue was with the way the NIC was processing incoming packets. Specifically, it was performing segmentation offloading, which reassembles incoming traffic into "super-packets" so that fewer (albeit larger) packets are passed up to the kernel. According to Bro documentation, the creation of these super-packets introduced artificial gaps between the packets of traffic that arrived to Bro [48]. Thus, Bro read these artificial gaps as legitimate traffic gaps. For these reasons, we turned off all NIC offloading functions with ethtool, which include:

```
tcp_segmentation offloading (TSO)
udp_fragmentation offloading (UFO)
generic segmentation (GSO)
generic receive offloading (GRO)
```

Disabling Checksum Validation (Bro)

The second issue had to do with the way Bro processes TCP headers. By default, Bro validates incoming TCP checksums - if the checksum is invalid, Bro discards it from its analysis [49]. This is problematic for analyzing locally generated traffic when checksum validation is offloaded to the network adapter (i.e. our case). In this case, all incoming packets have invalid checksums, as they are supposed to be calculated at their respective destinations, not our network adapter. Consequently, Bro would see these and get confused when reconstructing traffic traces and consequently discard all packets with invalid checksums. This was solved by adding a line to one of Bro's configuration files (/share/site/local.bro) that turns off checksum validation.

Disabling segmentation offloading and checksum validation led to an immediate improvement in CaptureLoss, as shown in the Figure 4.8

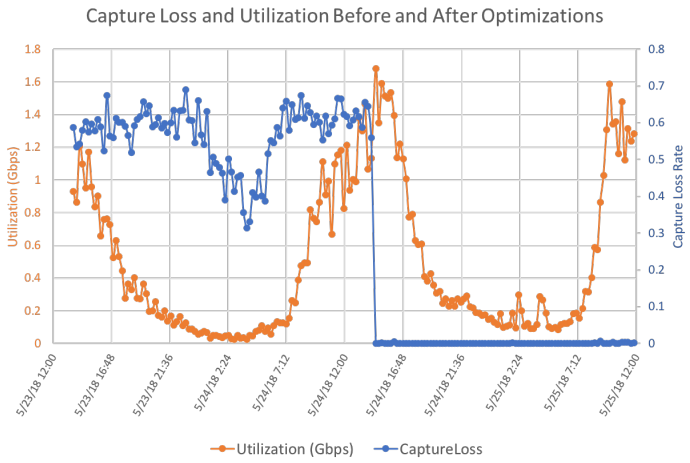

FIGURE 4.8: CaptureLoss Before and After Turning Off Offloading and Checksums

4.3.5 Summary

All the optimizations we applied are summarized in Figure 4.9. Combined, these optimizations led to a dramatic improvement with respect to packet capture. Before the optimizations, Bro dropped around 80% of incoming traffic, resulting in CaptureLoss rates as high as 95%. After the optimizations, Bro dropped less than 1% of traffic. We have seen consistently strong performance since then at speeds up to 6 Gbps with CaptureLoss rates consistently less than 3-4%. This gave us the confidence we needed to move forward with the rest of the pipeline.

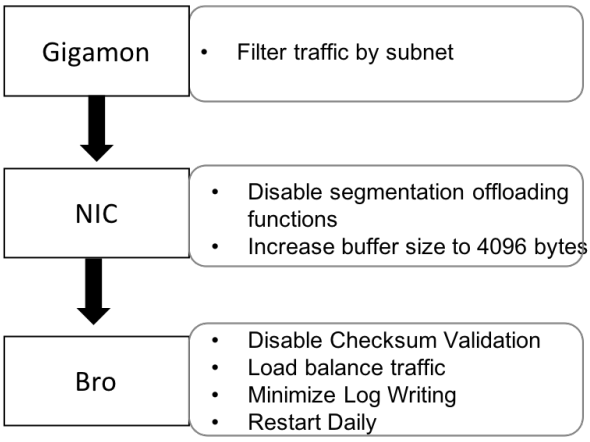

FIGURE 4.9: Summary of All Optimizations

4.4 Log Anonymization

When analyzing traffic at the University level, it is imperative to preserve user privacy. As such, we anonymize all UVA IP addresses using an algorithm called CtyptoPAn. (As a disclaimer, I did not work on the anomyzation component personally; this work was done by Curtis Khan, Will Hawkins and Alastair Nottingham. However, it is a crucial part of our system, so I've included a brief explanation of how it works to give the reader a full picture of the pre-processing pipeline.) Developed at Georgia Tech, CryptopAn is a prefix-preserving IP anonymization protocol. It is strongly founded in cryptographic theory and guarantees a unique one-to-one mapping between original and anonymized IP addresses. The proofs of this algorithm are beyond the scope of this thesis, but for more information, see [46]. The implications of these properties are two-fold: first, all behavior associated with an anonymized IP address originates from a single IP address. Second, if two IP addresses are on the same network, their anonymized counterparts will also be on the same network because their prefixes are preserved. Collectively, these properties allow us carry out our analysis without infringing on user privacy.

In our pipeline, anonnymization is outsourced to a separate machine called Ivy VM to reduce the strain on Ivy Bulwark (see Figure 4.1). After all traffic has been transferred to this machine, the Cryptopan algorithm is invoked to anonymize all observed UVA IP addresses in the conn logs, while all external IP addresses are preserved. Finally, the anonymized output is saved to a new directory and transferred to a machine on Rivana for traffic labeling.

4.5 Conclusion

This chapter has presented the components of our traffic processing pipeline. We have built a pipeline that captures raw traffic at speeds up to (theoretically) 10 Gbps with minimal packet drops and CaptureLoss. Additionally, the traffic is anonymized in such a way that we can still perform aggregations on the data while protecting user privacy. The success of this pipeline is largely due to the optimizations we made to the NIC, Bro, and the Ivy environment. Without them, Bro could only handle

speeds less than 1 Gbps. We measured the impact of each optimization with respect to packet drop rate and CaptureLoss. We reduced packet loss by load balancing traffic, reducing log writing, applying subnet filters, and restarting Bro on a daily basis. The key to reducing CaptureLoss turned out to be disabling RX offloading at the NIC and checksum offloading in Bro. While many of these optimizations are specific to Bro and our network configuration, the principles should apply to any high speed capturing environment. Chapter 5 details the next phase of the pipeline which involves labeling collected traffic.

Chapter 5

UVA Traffic Labeling Pipeline

After traffic is anonymized and pre-processed, it is sent to a machine on the Rivanna HPC for labeling. During this process, each connection is queried against a database containing data from a variety of feeds which are updated on a daily basis. The connection is assigned a vector of binary labels, with 1 indicating a match for each category in the database. In our case, each feed belonged to one of three categories: honeypots, blacklists, or whitelists. However, the system is highly flexible and can be configured to pull from an arbitrary number of feeds associated with arbitrarily many categories. In this sense, the system can be configured for any task which involves labeling IP addresses in network traffic via continuously updated feeds.

This chapter is laid out as follows. Section 5.1 describes each component of the labeling system in detail. Section 5.2 demonstrates how the system can be used in practice, describing the categories we use and the feeds we are pulling in our pipeline. Section 5.3 contains experimental results evaluating the utility of the generated labels based on a sample of labeled traffic. Finally, Section 5.4 concludes the chapter and suggests areas for future improvement.

5.1 Labeling System Architecture

Before describing each component of the system, I will formalize a few key notions. A *feed* is a list of known addresses associated with a certain activity or type of traffic. We define this activity or traffic type as the feed's *category*. The addresses on the list can take on a variety of forms, including domain names, URLs, or IP addresses. We refer to the systems using these addresses as *entities*, regardless of type, as they are

ultimately tied to an actor or being behind a screen or server, and it is relatively easy to switch from one representation to another.

An IP address belongs to a subnet (routing prefix), or a network containing a range of similar IP addresses. Subnets belong to a larger group of routing prefixes called an Autonomous System (AS), which is uniquely identified by its Autonomous System Number (ASN). An AS is a part of the internet that is administered by a single organization or enterprise (e.g. UVA or Google). Combined, the IP address, Subnet and ASN provide a holistic view of how an entity uses and connects to the internet.

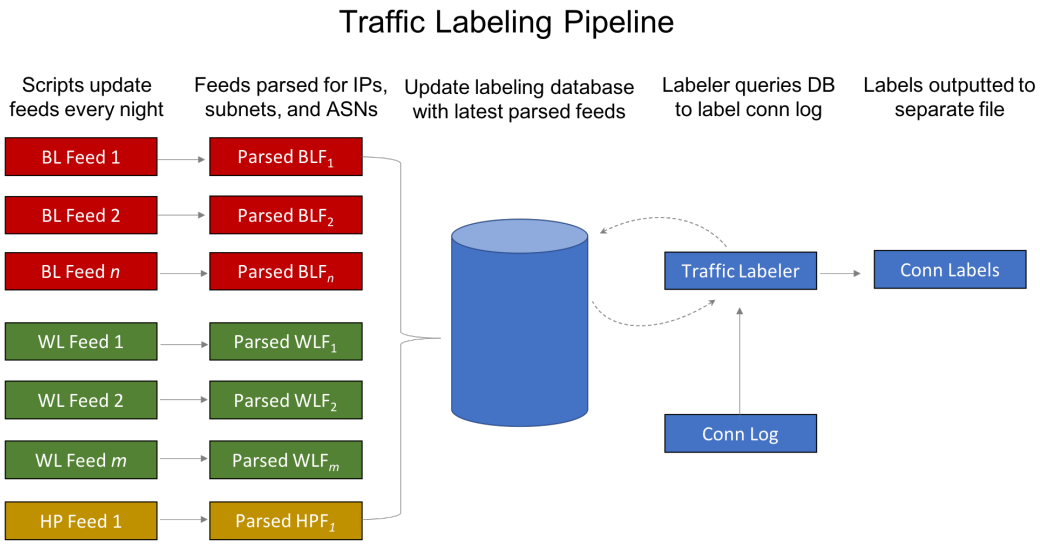

FIGURE 5.1: An overview of the labeling pipeline

Figure 5.1 gives a high-level overview of the labeling system. The process can be broken down into four steps: pull, parse, update and label. Each is described below.

1. *Pull*

   First, the feeds are pulled from their respective sources. This is performed with a series of shell scripts (one per feed) that download raw lists directly from the feed websites. Additionally, the latest ASN and subnet data are downloaded from routeviews.org, which contains snapshots of the latest subnet to ASN mappings used across the internet. An excerpt from the Abuse Ransomware feed and an ASN mapping file are shown in Figure 5.2.

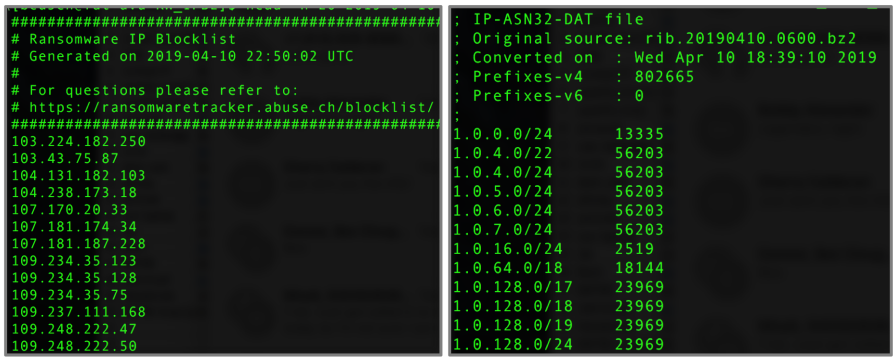

FIGURE 5.2: Samples of Abuse Ransomware Feed and routeviews file from Apr 10th, 2019

2. *Parse*

   Next, each feed is parsed for entity information which is resolved to domain name, IP address pairs. Parsing was done with Python, and the methods we used to convert between URLs, domain names, and IP addresses are summarized in figure 5.3. If the entity started as a URL, the `tldextract` library is used to obtain the domain name. From there, the socket library is used to resolve the domain name to an IP address. Finally, the IP address's subnet and ASN are found using pyasn's lookup function. Specifically, pyasn constructs a radix tree from the latest subnet-ASN mappings downloaded from routeview.org and the IP is resolved to a subnet, ASN pair by traversing this tree.

   Each node in the radix tree corresponds to a single subnet-ASN mapping. The nodes are organized hierarchically so that the subnets get smaller and smaller as one traverses the tree. The leaves are the smallest subnet-ASN mappings without any children. [38]. Since subnets are organized hierarchically, the traversal always results the longest prefix match. Thus, each IP gets mapped to at most one subnet and one ASN. Since the tree is built locally (i.e. no external dependencies) it yields much faster lookup times than online tools like whois. After this lookup process is complete, each entity's domain name, IP address, subnet, and ASN are added to a list of records which are exported to a CSV. Figure 5.4 demonstrates a real example from one of our blacklists.

   It should be noted for each entity, there's a chance it no longer resolves to an IP address. When this is the case, the entity is discarded, as there is no way

to leverage it for IP-based labeling. While attrition rates vary substantially by feed, our blacklist feeds suffered rates anywhere from 10-40%. This is not surprising as blacklists are known to grow stale over time [43].

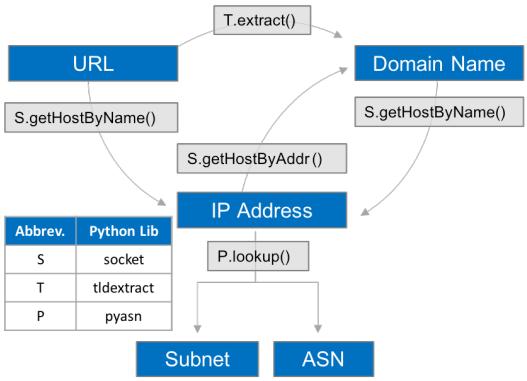

FIGURE 5.3: Converting Entities between URLs, Domain Names, IP Addresses, ASNs and Subnets in Python

```
INPUT: http://h-guan.com/djkbnnw.exe

In [11]: # 1) get domain name from URL
         res = tldextract.extract(url)
         domain = res.domain + '.' + res.suffix; domain

Out[11]: 'h-guan.com'

In [10]: # 2) get IP address from domain
         ip = socket.gethostbyname(domain); ip

Out[10]: '60.245.60.100'

 In [9]: # 3) get Subnet and ASN from IP address
         asndb = pyasn.pyasn('/Users/babraham/Google_Drive/Grad_School/Cyb
         asn, subnet = asndb.lookup(ip)
         print('subnet: {}, asn: {}'.format(subnet, asn))

         subnet: 60.245.56.0/21, asn: 9676

In [15]: print('OUTPUT: {}'.format([domain, ip, subnet, asn]))

         OUTPUT: ['h-guan.com', '60.245.60.100', '60.245.56.0/21', 9676]
```

FIGURE 5.4: Example of parsing process for a url from the Abuse-URL feed

3. *Update*

After all feeds are parsed, they are fed one-by-one into a centralized labeling database whose schema is shown in Figure 5.5 (Shout-out to Al Nottingham for helping with the design). The database contains a few useful constructs to store entity information. The first is a *MatchRecord*, or a tuple containing an IP address, domain name, a total hit count, and date. A MatchRecord says the IP, domain-name pair (X, Y) was seen on date D. The second, related construct

is an *Observation*. An observation links a *MatchRecord* to a particular feed. In other words, it says the IP domain-name pair (X, Y) was seen on date D *in feed* F. Say we are inserting record R containing Domain name D, IP address I, subnet S, ASN A, feed F, Category C and date DT into our database. Then the insertion algorithm is described in Figure 5.6

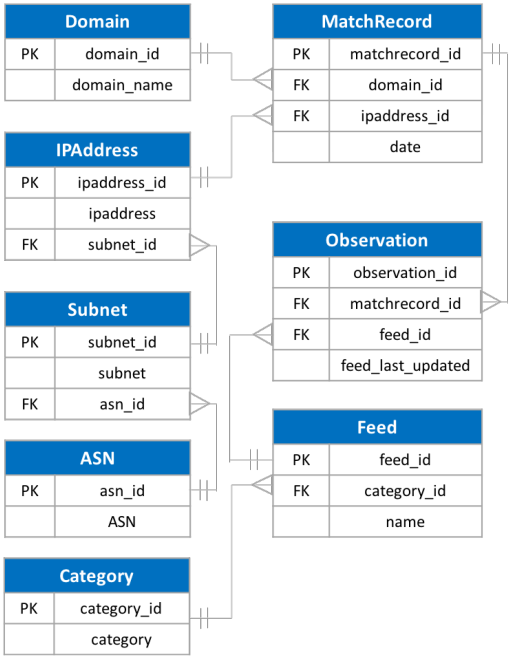

FIGURE 5.5: Schema for the Labeling Database

Insertion Algorithm

Let *R* = {*Domain*: D, *IP*: IP, *Subnet*: S, *ASN*: A, *Feed*: F, *Cat*: C, *Timestamp*: DT} and *X_id* be the database ID for element *X*.

1. if *C* is an unknown category, insert into **Category Table** and get its id.
2. if *F* is a new feed, insert record with *F* and *C_id* into **Feed Table**
3. Update **Domain Table**; create new Domain record if necessary
4. Update **ASN Table**; create new ASN record if necessary
5. Update **Subnet Table**. If no mapping between A and S, create entry in Subnet table
6. Update **IPAddress Table**. If *S_id* : *IP* pair not in table, create new entry.
7. Update **Matchrecord Table** with *D_id*, *IP_id* and *D*. Create new record if necessary.
8. Update **Observation Table** with MatchRecord ID, F_id, and Date. Create new record if necessary.

FIGURE 5.6: Algorithm for propogating new record information into database

4. *Label*

   Once the database is fully updated with the current date's feeds, it is used to label the current date's network traffic. The lookup process is summarized in Figure 5.7.

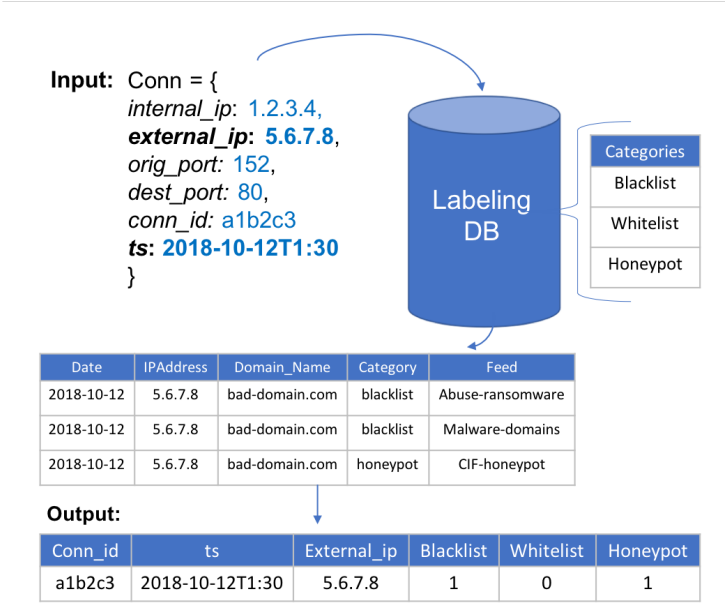

FIGURE 5.7: Process for labeling a connection with a database containing blacklist, whitelist, and honeypot data

Say we want to know about an external IP 5.6.7.8 on October 12^{th}. First, the database is queried for 5.6.7.8 and filtered by date. This is done by performing an inner join across all tables and filtering by the date column in the `MatchRecord` Table. If there is a match, the result of this query will be a table with IP, Subnet, ASN, Feed and Category information for date D. This information is then converted into a binary label vector which has one bit for every category in the database. For each category bit, 1 indicates a match while 0 indicates otherwise. This vector along with the connection ID, IP address, and timestamp are written out to a label file. After this process, each `conn` log for the current day will have a corresponding label file that can be easily merged to build a labeled dataset for supervised learning.

The above procedure works for both single IP lookups and bulk searchers. The only difference is in the number of IPs to filter by in the initial query. In our programs, we tend to use batch queries because each individual query is

expensive, requiring over five joins. It turns out if we instead load all the entity information into memory for a batch of IPs in a single shot, the lookup process is hundreds of times faster. In future work, we will investigate building an in-memory data structure from the database to perform real-time labeling. We will also investigate using HTTP host-names or SSL addresses instead of conn log IP addresses, as they are likely more reliable identifiers in the long run.

5.1.1 Design Considerations

While the above schema and insertion algorithms may seem overly complicated, they were deliberately designed this way to maximize flexibility while minimizing redundancy. The complexity arose from some intrinsic properties of network traffic which we had to account for in the database design. First, IP addresses are ephemeral. They are assigned to to an endpoint for a fixed lease, and are often allotted to other entities after the lease expires. Thus, a domain name could map to many different IP addresses throughout its lifetime, while a single IP address could be leased to many different domains over time. We accounted for this by creating the *MatchRecord* table, whose rows contain domain names, IP addresses and dates. If a domain name later gets a new IP addresses, a new *MatchRecord* is created containing the new IP-domain mapping and the future date.

Formally, the *MatchRecord* table serves as a bridge table between Domain Name and IP Address tables, creating a many-to-many relationship. Second, a given IP address can appear on multiple feeds in the same day. For this reason, we created the *Observation* table. As stated above, each observation links a *MatchRecord* to a particular feed. Thus, if an IP address appears on feeds A,B, and C that day, there will be three *Observations* mapping back to the same *MatchRecord* (i.e. a many-to-one relationship). Finally, the nature of the labeling task may change over time and require additional categories. We allowed for this by creating a separate Category table mapping each category to a unique ID. These IDs are auto-incremented integers, so each new category's ID is 1 greater than its predecessor.

The database was built using Python 2.7 and SQLite, a lightweight SQL client that allowed us to build the database locally. Because the entire database is stored

in a flat-file, it can easily be transferred to a new machine and used immediately so long as SQLite is supported.

5.2 Implementing the System for Our Research

The following sections describe how we're using the system in our research. We've populated the database with three types of feeds (categories): Honeypots, Blacklists, and Whitelists. We pull from 11 different feeds on a daily basis and leverage this data to label UVA network traffic. The coding for this project was mostly done in Python, SQL, and Bash and is located on Virginia Tech's GitLab repository [1].

5.2.1 Categories

Honeypots

As stated in Chapter 2, Honeypots serve as a simple yet effective way to detect threats on a network. We are currently running a Cowrie honeypot that is connected to a Community Honeypot Network (CHN) which is part of the Duke STINGAR Project [4] (Thanks to Jeff Collyer for integrating our honeypot to the STINGAR project). A small sample is shown in Figure 5.8. Currently, we are only leveraging the IPAddress and report time. However, future work should be done to investigate the utility of other fields like confidence, honeypot type, and hit count.

tlp	group	reporttime	ip address	firsttime	lasttime	count	tags	confidence	provider
green	everyone	2019-03-28T05:05	14.63.167.192	2019-02-20T22:42	2019-03-28T05:05	20	honeypot,cowrie	8	duke-chn
green	everyone	2019-03-28T05:07	131.255.91.30	2018-12-22T12:51	2019-03-28T05:07	313	honeypot,dionaea	8	duke-chn
green	everyone	2019-03-28T05:08	107.6.183.163	2019-03-28T05:08	2019-03-28T05:08	1	honeypot	8	uva-chn
green	everyone	2019-03-28T05:09	188.19.189.111	2019-03-28T05:08	2019-03-28T05:09	10	honeypot	8	uva-chn
green	everyone	2019-03-28T05:11	108.176.138.76	2019-03-28T05:02	2019-03-28T05:02	1	honeypot,dionaea	8	nccu-chn
green	everyone	2019-03-28T05:12	177.95.249.110	2019-03-28T05:02	2019-03-28T05:02	1	honeypot,dionaea	8	nccu-chn

FIGURE 5.8: Sample honeypot data in CIF log from Mar 28, 2019

The CHN allows multiple universities to pool resources and share threat intelligence across their networks. So if a threat is detected on Duke's honeypots, we will get that threat intelligence in our honeypot logs and vice versa. This communal approach dramatically increases the volume and signal-to-noise ratio of honeypot logs. We receive threat intelligence in the form of CIF logs, where each row is an observed threat originating from a specific public IP address. Thus, if we see any of

[1] https://code.vt.edu/p-core/connection-labeling

these IP addresses in our own network traffic, they are flagged as Honeypot traffic on that day.

Blacklists

Our Blacklist data comes from ten feeds that capture a broad spectrum of malicious traffic (each feed explained in detail below). Malicious feeds are pulled from the Abuse.ch project, MalwarePatrol, RiskAnalytics, and the Apache Foundation. Most feeds have a high degree of volatility, meaning that many entities are either added or removed from the lists on a daily basis. Thus, we pull the feeds daily and only use them to label traffic that occurred on the same day. However, the quality of these blacklists is still a lingering issue that will need to be dealt with in future work.

Whitelists

Finally, we pull a daily whitelist feed of top website visitation according to Cisco known as Cisco Umbrella. We implicitly use website popularity as a proxy for benign behavior. In general, this is an accurate substitute - sites like Google and Facebook are hard to compromise and thus genuinely indicate benign activity. However, the list fluctuates substantially over time, and it's possible for a malicious url to temporarily make the list until it is shut down. Thus, we only keep domains that have been on the list for at least a year. While we chose a year as a conservative estimate, future work will need to be done to figure out the optimal overlap threshold that maximizes coverage while minimizing false positives.

5.2.2 Feeds

Table 5.1 contains details about the feeds we've incorporated into the database. We currently collect ten different blacklist feeds, varying in scope and the behavior they track. Abuse Feodo, Abuse Zeus, MalwareDomains, and URLHaus contain IPs associated with malware, particularly C&C communication and malware distribution. Feeds like Zeus and Feodo track specific strains of malware, while others such as MalwareDomains and URLHaus are broader feeds, encompassing many different types of malware. Other blacklists focus on applications of malware such as spam

(spamassassin, mailwasher), ransomware (Abuse RansomWare), and SSL fingerprinting (Abuse-ssl). As described above, all whitelist data comes from AWS's Cisco Umbrella feed, and honeypot data comes from our Cowrie honeypot linked to the STINGAR honeypot network.

TABLE 5.1: Feeds Collected for Labeling Database

Feed	Organization	Category	Total # of Entities	Description
feodo	Abuse.ch	Blacklist	1.9k	Tracks all known C&C servers associated with the Feodo botnet
ransomware	Abuse.ch	Blacklist	330	Tracks ransomware IPs used for C&C servers, distribution sites, as well as payload sites
malware-domains	RiskAnalytics	Blacklist	27k	Tracks domains known to propagate malware and spyware
zeus	Abuse.ch	Blacklist	120	Tracks all known C&C servers associated with the Zeus botnet
ssl	Abuse.ch	Blacklist	110	Tracks malicious SSL certificates via sha-1 fingerprinting
url-haus	Abuse.ch	Blacklist	65k	Tracks urls being used for malware distribution
mailwasher	MalwarePatrol	Blacklist	27k	blocklist used by MailWasher Windows client to filter spam emails
firekeeper	MalwarePatrol	Blacklist	1.8k	Malware blocklist used by Firekeeper, a Firefox IDS
mozilla-adblock	MalwarePatrol	Blacklist	1.3k	Malware domains blocked by Mozilla adblock
spamassassin	Apache Foundation	Blacklist	1.7k	Tracks spam domains, scoring them based on email content
umbrella	Cisco	Whitelist	1 mil	Top 1 million domains visited in Umbrella global network
chn-cif	STINGAR	Honeypot	8-10k	honeypot network across universitites with pooled threat intelligence

5.3 Exploratory Analysis

To get a feel for the utility of these labels, we performed exploratory analysis on a random sample of labeled traffic. (**NOTE:** While the results below were obtained

from a different dataset, the methodology was first performed by my cyber capstone team). The traffic used for this experiment came from the UVA network on Jan 28, 2019. The labels were generated using the labeling system and procedures described above and were joined with the connection logs by connection ID and timestamp. A random sample of 100,000 connections was then pulled from the dataset. Each connection then had a binary label vector containing honeypot, blacklist, and whitelist flags. Since we are often interested in binary classification, these labels were simplified into malicious and benign using the rules in Table 5.2.

TABLE 5.2: Label Mappings

Whitelist	Blacklist	Honeypot	Ensemble Label
1	0	0	Benign
0	1	0	Malicious
0	0	1	Malicious
0	1	1	Malicious
0	0	0	Unknown
1	1	0	Unknown
1	0	1	Unknown
1	1	1	Unknown

Overall, 71% of the traffic was 'Unknown', 19% was 'Benign', and 10% was 'Malicious'. All 'Unknown' combinations were discarded from this analysis for simplicity and treated the final labels as cluster assignments. As a benchmark, we applied an anomaly detection algorithm called Isolation Forest to the same samples but without their labels. Connections labeled as 'anomalous' by the isolation forest were then assumed to be malicious, while all non-anomalous labels were assumed to be benign.

5.3.1 Cluster Analysis

We evaluated the separation between the clusters using a distance measure called the Silhouette Score. This score is a reliable evaluation metric that measures the efficacy of a clustering algorithm by computing pointwise distance between the centroids of each cluster. Formally, it is defined as

$$S = \frac{a - b}{\max(a, b)}$$

where a is the mean distance between a sample and all other points in the same class and b is the mean distance between a sample and all other points in the next nearest cluster. Possible coefficient values span -1 and 1. Scores close to zero indicate overlapping clusters, -1 indicates incorrect clustering, and 1 indicates dense, correct clustering. The Silhouette score of the ensemble labels was .71, while the Isolation Forest achieved a score of only .52. This suggests that the ensemble labels create superior clusters to those of the Isolation Forest.

5.3.2 Port Analysis

As a second experiment, we compared the port distributions of malicious and benign traffic. Since ports are inherently tied to activities on the Internet, and most benign traffic occurs on a handful of well-defined ports, we'd expect malicious and benign traffic to have different port distributions. In order to test this hypothesis, we performed the chi-square test on the class's port distributions. The test yielded a p-value of 0.0001, suggesting that the malicious and benign port counts did not come from the same distribution. This further reinforces that the labels are picking up on legitimate behaviors.

5.3.3 Geolocation Analysis

Finally, we compared the geolocation distributions of malicious and benign traffic. We were particularly interested in the fraction of connections originating in the US for each class. Intuitively, we'd expect most benign connections to an American University to originate from the States. Conversely, we'd expect most suspicious activity to come from abroad (i.e. Russia, China). Table 5.3 shows the portion of US connections by class for the Ensemble labels and Isolation Forest labels. As expected, only 8% of malicious ensemble connections come from within the US, while 85% of benign connections originate from American soil. Interestingly, the opposite seems apply to the Isolation Forest labels - most anomalies occurred within the US, while most non-anomalies occurred outside the US.

TABLE 5.3: Portion of Connections from the US By Class

Labeling Scheme	Malicious	Benign
Ensemble	.11	.80
Isolation Forest	.86	.22

5.3.4 Conclusions

Our experimental results suggest that the labels produced by the pipeline both statistically and intuitively make sense. They produce significantly distinct clusters (Silhouette Score of .64), the port distributions between classes are fundamentally different (p=.0001), and the classes are geographically distributed in a way that we'd expect. These results also suggest that unsupervised learning algorithms cannot pick up on the inherent patterns derived from our supervised labels. Ultimately, these results give us confidence to leverage these labels for supervised learning in the future.

5.4 Future Improvements

While the system described above is by all means functional, we plan to improve it in many ways. First, we will incorporate more blacklist and whitelist feeds into the labeling process. Each new feed increases our labeling coverage and allow us capitalize on new pockets of the internet we previously had no visibility into. Second, we will start tracking how frequently each IP address in the database appears in our traffic. This will be done by populating the `total_hits` field during the labeling process. These hit counts will tell us how 'useful' each feed is for labeling purposes. Finally, we plan to incorporate attack simulation data into the labeling pipeline. We have developed the ability to perform isolated, IP-based attacks on our network, so any flows associated with these attacks could conceivably be labeled with our database if we know the IP addresses of those involved and the timing of the attack.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

This thesis presented a network traffic processing pipeline and modeling techniques that can be used to build an anomaly-based intrusion detection system. Before our traffic pipeline was in place, we conducted a pilot study which explored machine learning techniques for botnet detection on third party data. After completing this study, we turned our attention to building out and fine-tuning our own traffic collection pipeline which required many different optimizations. Finally, we presented an autonomous traffic labeling system that can perform IP-based traffic labeling. The key findings and conclusions from each chapter are summarized below.

Chapter 3 presented the results of our botnet detection study. The goal of this study was to not only to find the best algorithm botnet detection, but also to discern useful features for discriminating between malicious and benign traffic. We found that statistics pertaining to source packet counts, time intervals, and certain flag counts had the most predictive power. Of all tested models, Random Forest performed the best when evaluated with traditional cross validation, achieving an F1 score of .99. However, traditional CV fails to capture the ability to generalize to novel botnet types. To address this, we developed a scheme called LOBO-CV, which involves training the model on all but one family's traffic and evaluating it's ability to discriminate between the left out family's traffic and benign traffic. When evaluated with this scheme, all model performances dropped and varied significantly from family to family. However, ensembling predictions from our top two models (Random Forest and Logistic Regression) greatly improved LOBO-CV performance

and consequently the model's ability to generalize to novel, unseen attack types.

The contributions of this study are three-fold. First, our results corroborate previous work in the field which suggests Random Forest is a strong model for anomaly detection [19, 25, 11]. Second, we introduce LOBO-CV, a novel evaluation technique that captures generalizability better than traditional K-fold CV. While this scheme was built with botnet detection in mind, the principle can apply to any machine learning problem where the target class is composed of smaller subsets that change over time. Finally, we demonstrated that ensembling two distinct model classes can further improve generalizability, and in some cases improve on both individual model's performance.

Chapter 4 detailed our traffic collection pipeline, which processes raw traffic at speeds up to (theoretically) 10 Gbps with minimal packet drops and capture loss. Additionally, the traffic is anonymized in such a way that we can still perform aggregations on the data while protecting user privacy. The success of this pipeline is largely due to the optimizations we made to the NIC, Bro, and the Ivy environment. The goal of these optimizations was to minimize packet drops and capture loss as much as possible. The most effective measures to counteract packet drops proved to be load balancing traffic, reducing log writing, applying subnet filters, and restarting Bro on a daily basis. The keys to reducing capture loss turned out to be disabling RX offloading at the NIC and checksum offloading in Bro. The takeaway from this process is that building an effective traffic capturing pipeline requires a holistic approach and a nuanced understanding of the system, because there are so many places along the way where things can go wrong.

Chapter 5 presented a system for labeling network traffic based on IP address and time. The system consists of a centralized labeling database that pulls from each feed on a daily basis. The system automatically configures itself to run in the user's environment, and as such requires minimal overhead. It is highly flexible and works for any number of categories and any number of feeds. Consequently, it can be extended to any IP-based traffic labeling applications.

In our research however, we tailored the pipeline to blacklist, whitelist, and honeypot labeling. The database pulled from ten different blacklist feeds, a whitelist feed, and a honeypot feed. We evaluated the utility of our labels by conducting

EDA on a sample of labeled data. Our experimental results suggest that the labels both statistically and intuitively make sense. They produce significantly distinct clusters (Silhouette Score of .64), the port distributions between malicious and benign traffic are fundamentally different (p=.0001), and the classes are geographically distributed in a way that we'd 'expect. In each case, the ensemble labels achieved better results than the benchmark anomaly detection algorithm. This suggests that unsupervised anomaly detection algorithms cannot infer the same type of clustering derived from our labels. This raises the question of whether all malicious traffic are actually anomalies. Our results certainly suggest that the 'anomalies' in the data had little overlap malicious traffic. Perhaps we observe this because hackers often deliberately mimic benign traffic when designing and executing their malware. Thus from a statistical perspective, a large portion of their traffic would look benign. Whether or not this is true of malicious traffic on our network has yet to be determined. However, this phenomenon is well-documented in the field at large and is one of the main reasons why attack detection is such a challenging problem.

6.2 Limitations and Future Work

There are definitely some limitations, and consequently, avenues of future work to pursue in this thesis. The first would be improving the evaluation technique we used in our botnet detection study. While our results are encouraging, they might not generalize well to real network traffic, as our datasets had a balanced distribution of malicious and benign traffic in both the training and test sets. In the wild, malicious traffic usually constitutes a small fraction of the entire dataset. This class imbalance will prove to be a challenge for traditional supervised learning models and will likely require a combination of cost-sensitive learning and sampling techniques. We are currently struggling with this issue in a capstone project - we can accurately detect malicious traffic in an evenly split sample, but the false positive rate increases as we add more benign traffic.

With respect to traffic labeling, the main challenge at the moment is the lack of coverage. Empirical data suggests we can only truly label 20% of our traffic at best with this approach. One of the key reasons for this is because a large portion of

popular (benign) sites are hosted by Content Delivery Networks (CDNs) which obfuscate the true IP address of a hosted destination. Specifically, when a user goes to website X hosted behind a CDN, it must pass through the CDN proxy first. Consequently, as the traffic leaves our network, the destination IP address points to the CDN routers, not the website itself. Thus, we see substantial traffic to CDNs that is likely being re-routed to popular sites. One might suggest obtaining the IP space for these CDNs and labeling all traffic in those networks as benign. However, malicious actors can infiltrate websites behind CDNs, so there is no way to know for sure if the traffic is benign.

One way to improve coverage is to leverage other entity metadata, such as its subnet or ASN. Instead of using the feeds to extract IPs, we can use them to characterize behavior of subnets and ASNs. The idea to model the distribution of malicious and benign IP addresses of an ASN or subnet across all feeds. This way, even if there is no record of a particular IP in the database, we can still glean information about its likelihood of being malicious from its subnet or ASN. This information is tracked in our labeling database, but our initial results using this data have been mixed.

While these changes will greatly improve the system's performance and utility, the labeling process is still ultimately retrospective and performed in batches. The long term vision for the pipeline is to label traffic real-time as it comes in off the wire. Theoretically, this can be achieved by switching to a streaming service like Kafka. In such an architecture, each new connection would be sent from Bro to Kafka as a data object instead of being written to hourly flat files. These objects would then be labeled by querying the SQL database in real time and subsequently exported to consumers. However, this approach would require substantial retooling of our current pipeline, and it's gains have yet to be quantified or ascertained. Ultimately, the added capability of streaming services will have to be weighted against the additional complexity incurred by switching to real-time traffic labeling.

Bibliography

[1] URL: https://stratosphereips.org/category/dataset.html.

[2] URL: https://www.zeek.org.

[3] URL: https://www.bro.org/sphinx/scripts/policy/misc/capture-loss.bro.html.

[4] URL: https://stingar.security.duke.edu/about-2/.

[5] *10 top network intrusion detection tools for 2018*. 2019. URL: https://www.comparitech.com/net-admin/network-intrusion-detection-tools/#Signature-based_IDS.

[6] Manos Antonakakis et al. "Building a Dynamic Reputation System for DNS". In: *Proceedings of the 19th USENIX Conference on Security*. USENIX Security'10. USENIX Association, 2010, pp. 18–18. ISBN: 888-7-6666-5555-4. URL: http://dl.acm.org/citation.cfm?id=1929820.1929844.

[7] Shehar Bano. "A Study of Botnets: Systemization of Knowledge and Correlation-based Detection". In: (2012). URL: http://sheharbano.com/assets/publications/ms_thesis_sheharbano.pdf.

[8] Steven Andrew Barker. "Comparison of Ring-Buffer-Based Packet Capture Solutions". In: (2015). DOI: 10.2172/1225853.

[9] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[10] *Bro mailing list: Question about capture loss script vs. broctl netstats*. 2013. URL: https://bro.bro-ids.narkive.com/BVEU5gCX/question-about-capture-loss-script-vs-broctl-netstats.

[11] A. L. Buczak and E. Guven. "A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection". In: *IEEE Communications Surveys Tutorials* 18.2 (2016), pp. 1153–1176. ISSN: 1553-877X. DOI: 10.1109/COMST.2015.2494502.

[12] Tao Cai and Futai Zou. "Detecting HTTP botnet with clustering network traffic". In: *Wireless Communications, Networking and Mobile Computing (WiCOM), 2012 8th International Conference on*. IEEE. 2012, pp. 1–7.

[13] *Cyber Crime Costs $11.7 Million Per Business Annually*. URL: https://www.securitymagazine.com/articles/88338-cyber-crime-costs-117-million-per-business-annually.

[14] "Cyberattack hits UVa". In: *The Daily Progress* (2015). URL: http://www.dailyprogress.com/news/cyberattack-hits-uva/article_59b0454c-42c7-11e5-88ae-53d47ac2265c.html.

[15] Crovella Diot and Lakhina. "Mining Anomalies Using Traffic Feature Distributions". In: (2005).

[16] Meisam Eslahi, Habibah Hashim, and NM Tahir. "An efficient false alarm reduction approach in HTTP-based botnet detection". In: *Computers & Informatics (ISCI), 2013 IEEE Symposium on*. IEEE. 2013, pp. 201–205.

[17] Meisam Eslahi et al. "Periodicity classification of HTTP traffic to detect HTTP Botnets". In: *Computer Applications & Industrial Electronics (ISCAIE), 2015 IEEE Symposium on*. IEEE. 2015, pp. 119–123.

[18] Farhood Farid Etemad and Payam Vahdani. "Real-time botnet command and control characterization at the host level". In: *Telecommunications (IST), 2012 Sixth International Symposium on*. IEEE. 2012, pp. 1005–1009.

[19] A. Nur Zincir-Heywood Malcolm I. Heywood Fariba Haddadi Dylan Runkel. "On Botnet Behaviour Analysis using GP and C4.5". In: *GECCO Comp '14 Proceedings of the Companion Publication of the 2014 Annual Conference on Genetic and Evolutionary Computation* (2014), pp. 1253–1260.

[20] *February 2019 Web Server Survey*. 2019. URL: https://news.netcraft.com/archives/category/web-server-survey/.

[21] Pavel Filonov, Andrey Lavrentyev, and Artem Vorontsov. "Multivariate Industrial Time Series with Cyber-Attack Simulation: Fault Detection Using an LSTM-based Predictive Data Model". In: *CoRR* abs/1612.06676 (2016). eprint: 1612.06676. URL: http://arxiv.org/abs/1612.06676.

[22] Giovanni Vigna Christopher Kruegel Florian Tegeler Xiaoming Fu. "BotFinder: Finding Bots in Network Traffic Without Deep Packet Inspection". In: *CoNEXT '12 Proceedings of the 8th international conference on Emerging networking experiments and technologies* (2012), pp. 349–360.

[23] S. Gallenmüller et al. "Comparison of frameworks for high-performance packet IO". In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. 2015, pp. 29–38. DOI: 10.1109/ANCS.2015.7110118.

[24] Martin Grill and Martin Rehák. "Malware detection using HTTP user-agent discrepancy identification". In: *Information Forensics and Security (WIFS), 2014 IEEE International Workshop on*. IEEE. 2014, pp. 221–226.

[25] Fariba Haddadi et al. "Botnet behaviour analysis using ip flows: with http filters using classifiers". In: *Advanced Information Networking and Applications Workshops (WAINA), 2014 28th International Conference on*. IEEE. 2014, pp. 7–12.

[26] *https://cran.r-project.org/web/packages/caret/index.html*.

[27] ANTON SCHWAIGHOFER NEIL D. LAWRENCE JOAQUIN QUIÑONERO-CANDELA MASASHI SUGIYAMA. *Dataset Shift in Machine Learning*. Massachusetts Institute of Technology, 2009.

[28] Michael Kuhl et al. "Cyber attack modeling and simulation for network security analysis". In: 2008, pp. 1180–1188. ISBN: 978-1-4244-1306-5. DOI: 10.1109/WSC.2007.4419720.

[29] Berkley Labs. "100G Intrusion Detection". In: (2015). URL: https://www.cspi.com/wp-content/uploads/2016/09/Berkeley-100GIntrusionDetection.pdf.

[30] William Robertson Engin Kirda Christopher Kruegel Leyla Bilge Davide Balzarotti. "DISCLOSURE: Detecting Botnet Command and Control Servers Through Large-Scale NetFlow Analysis". In: *ACSAC '12 Proceedings of the 28th Annual Computer Security Applications Conference* (2012), pp. 129–138.

[31] Ke Li, Chaoge Liu, and Xiang Cui. "POSTER: A Lightweight Unknown HTTP Botnets Detecting and Characterizing System". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2014, pp. 1454–1456.

[32] J. Liang, L. Sankar, and O. Kosut. "Vulnerability Analysis and Consequences of False Data Injection Attack on Power System State Estimation". In: *IEEE Transactions on Power Systems* 31.5 (2016), pp. 3864–3872. ISSN: 0885-8950. DOI: 10.1109/TPWRS.2015.2504950.

[33] Chen Lu and Richard Brooks. "Botnet traffic detection using hidden markov models". In: *Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research*. ACM. 2011, p. 31.

[34] Leigh Metcalf and Jonathan M. Spring. "Blacklist Ecosystem Analysis". In: *Proceedings of the 2nd ACM Workshop on Information Sharing and Collaborative Security - WISCS 15* (2015). DOI: 10.1145/2808128.2808129.

[35] Marcin Nawrocki, Matthias Wallhisch, and Thomas Schmidt. "A Survey on Honeypot Software and Data Analysis". In: (2016). DOI: 1608.06249. URL: https://arxiv.org/pdf/1608.06249.pdf.

[36] A. Oprea et al. "Detection of Early-Stage Enterprise Infection by Mining Large-Scale Log Data". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 2015, pp. 45–56. DOI: 10.1109/DSN.2015.14.

[37] Michael Purzynski and Peter Manev. *Suricata Extreme Performance Tuning*. 2019.

[38] *pyasn Documentation*. URL: https://pypi.org/project/pyasn/.

[39] P&S Marketing Research. "Artificial Intelligence (AI) in Cyber Security Market by Service Type Demand Forecast, 2013-2023". In: (2017). URL: https://www.researchandmarkets.com/research/35scht/global_artificial?w=5.

[40] J.Stiborek A.Zunino S.García M.Grill. "An empirical comparison of botnet detection methods". In: *Computers & Security* (2014), pp. 100–123.

[41] Jungsuk Song et al. "Statistical Analysis of Honeypot Data and Building of Kyoto 2006+ Dataset for NIDS Evaluation". In: *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*. BADGERS '11. ACM, 2011, pp. 29–36. ISBN: 978-1-4503-0768-0. DOI: 10.1145/1978672.1978676. URL: http://doi.acm.org/10.1145/1978672.1978676.

[42] B Soniya and M Wilscy. "Using entropy of traffic features to identify bot infected hosts". In: *Intelligent Computational Systems (RAICS), 2013 IEEE Recent Advances in*. IEEE. 2013, pp. 13–18.

[43] Matija Stevanovic et al. "On the ground truth problem of malicious DNS traffic analysis". In: *Computers & Security* 55 (2015), 142–158. DOI: 10.1016/j.cose.2015.09.004.

[44] Pablo Torres et al. "An analysis of Recurrent Neural Networks for Botnet detection behavior". In: *Biennial Congress of Argentina (ARGENCON), 2016 IEEE*. IEEE. 2016, pp. 1–6.

[45] Binbin Wang et al. "Modeling connections behavior for web-based bots detection". In: *e-Business and Information System Security (EBISS), 2010 2nd International Conference on*. IEEE. 2010, pp. 1–4.

[46] Jun Xu et al. "Prefix-preserving IP address anonymization: measurement-based security evaluation and a new cryptography-based scheme". In: *10th IEEE International Conference on Network Protocols, 2002. Proceedings.* (2004). DOI: 10.1109/icnp.2002.1181415.

[47] Daniel Zammit. "A machine learning based approach for intrusion prevention using honeypot interaction patterns as training data". PhD thesis. 2016. DOI: 10.13140/RG.2.1.1996.0561.

[48] *Zeek Frequently Asked Questions*. URL: https://www.zeek.org/documentation/faq.html#how-can-i-reduce-the-amount-of-captureloss-or-dropped-packets-notices.

[49] *Zeek Frequently Asked Questions*. URL: https://www.zeek.org/documentation/faq.html#why-isn-t-zeek-producing-the-logs-i-expect-a-note-about-checksums.