8-2017

# Operating System Identification by IPv6 Communication using Machine Learning Ensembles

Adrian Ordorica
*University of Arkansas, Fayetteville*

Operating System Identification by IPv6 Communication using Machine Learning Ensembles

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

by

Adrian Ordorica
University of Arkansas
Bachelor of Science in Computer Science, 2015

August 2017
University of Arkansas

This thesis is approved for recommendation to the Graduate Council.

_____
Dale R. Thompson, Ph.D., P.E.
Thesis Director

_____
Jia Di, Ph.D.
Committee Member

_____
Qinghua Li, Ph.D.
Committee Member

## ABSTRACT

Operating system (OS) identification tools, sometimes called fingerprinting tools, are essential for the reconnaissance phase of penetration testing. While OS identification is traditionally performed by passive or active tools that use fingerprint databases, very little work has focused on using machine learning techniques. Moreover, significantly more work has focused on IPv4 than IPv6. We introduce a collaborative neural network ensemble that uses a unique voting system and a random forest ensemble to deliver accurate predictions. This approach uses IPv6 features as well as packet metadata features for OS identification. Our experiment shows that our approach is valid and we achieve a neural network ensemble average accuracy of 85% over 100 sets of neural networks with a highest accuracy of 96%. Furthermore, we explore the impact of additional training for poor neural network accuracy, and we show that our system can achieve an average accuracy of 93%, which is an 8% improvement over the previous approach. A random forest of 30 decision trees attains an average accuracy of 93.6% and a best accuracy of 96% when given a dataset of Windows and Linux packets. Finally, as packets from the Mac OS is introduced into the dataset, the random forested performed with an average accuracy of 89.6% and a best accuracy of 93.2%.

## ACKNOWLEDGEMENTS

I would like to thank Dr. Dale R. Thompson in the Department of Computer Science and Computer Engineering at the University of Arkansas. Dr. Thompson was very supportive through every step of the way. He steered me in the right direction and provided valuable guidance and encouragement to accomplish this thesis.

I would also like to thank Dr. Qinghua Li, Dr. Jia Di, and all the faculty who guided my development as a scientist, engineer, and professional.

Finally, I want to thank my significant other, Renee Jennings, for her love, continuous encouragement, and positivity that kept me motivated through the research and writing process.

## TABLE OF CONTENTS

## LIST OF FIGURES

## LIST OF TABLES

# 1. INTRODUCTION

## 1.1 Motivation

Internet Protocol version 6 (IPv6) is the most recent numbering system that provides more IP addresses than Internet Protocol version 4 (IPv4). The growing need for IPv6 is slow but inevitable with rising IP address consumption. The new address space uses eight sets of four hexadecimal addresses separated by a colon (:) like: xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx (x would be a hexadecimal value) providing up to $3.42 \times 10^{38}$ total addresses. IPv6 simplified header structures lead to faster routing compared to IPv4. Different operating systems (OS) have different implementations of IPv6 that exhibit slight variations in the protocol. These features can be used to do passive identification of the operating system, which is called OS identification.

OS identification is important to network security with its relationship to the reconnaissance phase of penetration testing. Knowing the OS is essential for attackers to accordingly use tools and programs when gaining access to their targets. The network layer of the Open Systems Interconnection (OSI) model does not contain any explicit information about the operating system of the network device generating traffic. However, certain features are unique to each operating system.

Machine learning focuses on the ability for computers to learn without being explicitly programmed. This is achieved with a combination of algorithms, simulated neural networks, and ensembles of learning methods for classification. These simulated neural networks are intertwined weights that adjust after passing in features (input) transformed by an activation function and taking the difference between the actual labels and the prediction (output). A greedy algorithm known as backpropagation provides a fast solution to pattern recognition in this

supervised learning experiment. Random forests provide an even faster solution that rely on a multitude of decision trees as an ensemble to make fast predictions with controlled variance.

Present passive OS identification methods match the gathered network traffic with previously developed IPv6 signature databases. This thesis is an extension to a previous work that applied neural networks to fingerprint IPv6 packets from Windows and Linux OSs [18]. The extensions to [18] include using decision trees, random forests, and Mac OS packets. The approach in this work is to use supervised machine learning techniques and random forests to learn the slight variations in the IPv6 network implementations of different OSs.

### 1.2 Objective

The objective of this work is to identify operating systems by passively observing the IPv6 protocol packets, unlike active techniques that send packets to the target system. This passive identification of OSs uses multiple machine learning algorithms such as neural networks and random forest to learn the implementation differences of the IPv6 protocol stacks in different OSs for identification. Then, algorithms are evaluated by their accuracy and the speed that they can process the packets.

### 1.3 Approach

The process is to identify candidate machine learning algorithms, gather IPv6 traffic from a network with computers running known operating systems, identify IPv6 features, and test/verify. First, the limits and capabilities of current machine learning algorithms are explored including neural networks, decision trees, and random forests. For example, different neural network configurations perform better with different problems, such as image recognition, pattern prediction, or robots that mimic actions observed and adjusts to various environments

[24, 7, 2]. Secondly, IPv6 traffic is gathered from a network with computers running known operating systems to provide a realistic dataset that contains IPv6 traffic. Then, IPv6 features are identified to determine how effective the features can be used to identify operating systems. Finally, part of the dataset is used to train the chosen machine learning algorithms and the remaining portion of the dataset is used to test the accuracy and speed of the machine learning algorithms.

### 1.4 Organization of this Thesis

The rest of the paper is organized as follows: Section 2 covers the foundation of IPv6 and the contrast to IPv4, as well as an overview of OS identification and machine learning techniques. The methods for performing passive OS identification are discussed in Section 3. Section 4 presents the results comparing a variety of setups. Conclusions and future work are discussed in Section 5.

# 2. BACKGROUND

## 2.1 Key Concepts

A basic understanding of the following topics is important to comprehend the work in Section 3. First, an overview of IPv6 is discussed and compared with IPv4. Then, we cover some basics on the Stateless Address Autoconfiguration mechanism. Afterwards, an overview of OS identification and fingerprinting is presented. Lastly, machine learning techniques used in this work are introduced.

### 2.1.1 Comparison of IPv4 and IPv6

While the IPv4 address space contains just over four billion unique addresses, the availability of these addresses is quickly becoming scarce. IPv6, the successor to IPv4, increases the address space to three hundred and forty undecillion (undecillion = $10^{36}$) addresses by changing the 32-bit address to a 128-bit address. Along with the larger addressing space, IPv6 brings other technical benefits. As shown in Figure 1, the format of the IPv6 header is designed to be simpler than the IPv4 header. The IPv6 header reduces the fourteen fields in the IPv4 header to eight fields. While seven fields are removed from the IPv4 header, the functionality of these fields are combined into the next header field of the IPv6 header. The next header field is optional, yet can chain additional headers if the packet requires additional options or information. Four of the field names and positions have changed and a new field, known as the flow label, is introduced. The flow label is a quality of service mechanism to avoid congestion of the network [1].

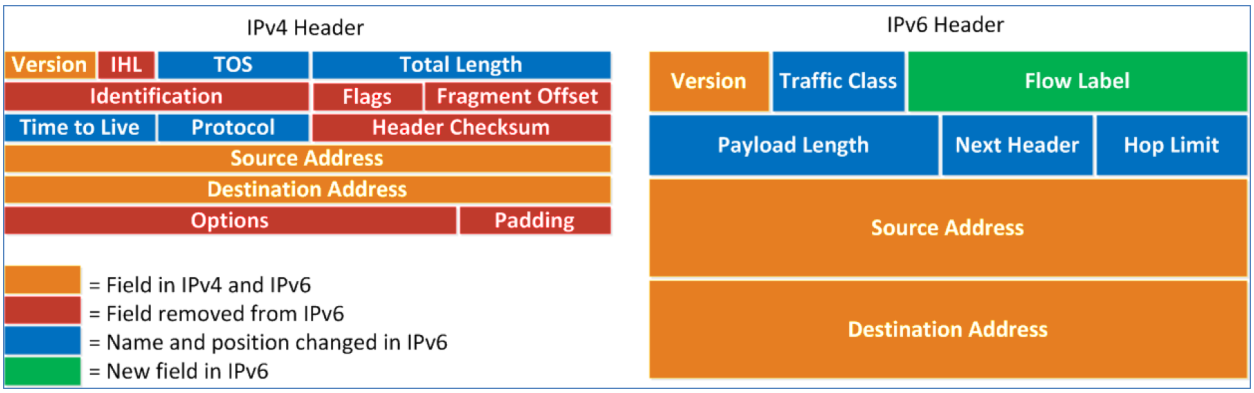


**Figure 1. IPv4 Header and IPv6 Header Comparison [23]**

### 2.1.2 Neighbor Discovery Protocol (NDP)

The Neighbor Discovery Protocol (NDP) is essential for any host to connect and to stay on a Local Area Network (LAN) when using IPv6. There are five Internet Control Message Protocol (ICMP) packet types: Router Solicitation (RS), Router Advertisement (RA), Neighbor Solicitation (NS), Neighbor Advertisement (NA), and Redirect. There is constant communication using these packets on the LAN for host machines to know its neighbors and routers.

As introduced in RFC 4862, IPv6 Stateless Address Autoconfiguration (SLAAC) is a mechanism that requires no manual configuration in hosts, minimal configuration for routers, and no additional servers [25]. SLAAC provides a new stateless method that off-loads router computing to the host. SLAAC relies on the NDP features such as Duplicate Address Detection (DAD), router discovery, prefix discovery, parameter discovery, address resolution, next-hop determination, and host reachability. The SLAAC process is shown in Figure 2.

When a new host uses SLAAC to join a network, a multi-cast NS packet containing a generated address is sent on the LAN. The solicitation message is used to determine whether any other host currently uses the address. If a host replies with a NA packet, then the new host

cannot use the proposed address and must try again. Otherwise, the proposed address is assigned to an interface. Then, a RS packet is sent on the LAN to discover any routers and request any necessary parameter or prefix information for generating network traffic. While a few packet header fields have some necessary values, there is some flexibility on a subset of IPv6 header fields. These field values are determined by the OS. While there is no harm in having different values for these header fields during implementation, it does give these packets unique characteristics.

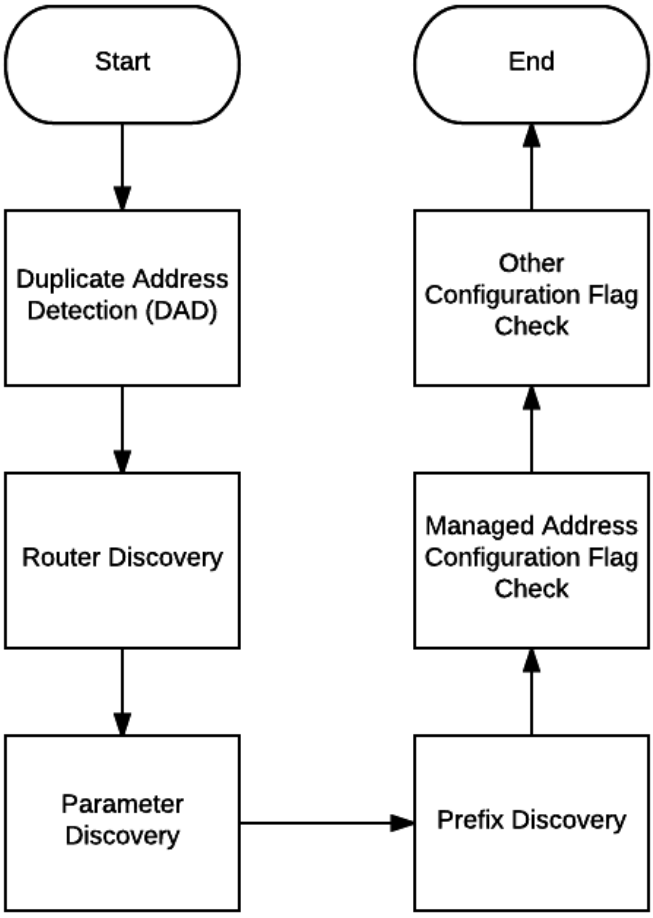


**Figure 2. SLAAC Process**

### 2.1.3 Operating System Identification

Operating system (OS) identification is important to penetration testing as this information can help the attacker predict known vulnerabilities. Any layer of the Open Systems Interconnection (OSI) model does not contain any explicit information about the operating system of the network device generating traffic. However, network tools can be used to distinguish OSs based on unique features within network packets. These OS identification tools can be categorized as either passive or active.

Passive OS identification tools perform a packet analysis on captured network traffic by looking for subtle differences between packets. The most popular passive tools include p0f, Satori, and NetworkMiner. These tools have several common techniques based on analyzing IP TTL/hop limit values, IP ID values, TCP window size, TCP options, DHCP requests, ICMP requests, and observing many other particular fields within a packet [15]. These tools take a combination of values to form a signature. Then, this signature is compared to a database that is constantly growing with OS signatures. These passive tools do not probe the victim host and do not generate network traffic. This can make for detection of passive OS identification very difficult.

Active OS identification tools probe the victim host and compare the response to a database. Active OS identification tools are usually quicker than passive OS identification tools as they can actively query the victim for packets, but can be easily detected as these active tools are generating network traffic. The most popular active OS identification tool is Nmap. Nmap sends a series of TCP and UDP packets to the victim host and examines the response to generate a signature. Like passive OS identification tools, the signature is compared to a database containing several thousand OS signatures.

### 2.1.4 Machine Learning

Machine learning focuses on the ability for computers to learn without being explicitly programmed. Through a combination of algorithms and data structures, machine learning has proven to be an effective method for classifications, as shown by [10, 14]. This work uses machine learning ensembles based on artificial neural networks and random forests. Artificial neural networks (ANNs) are intertwined weights that adjust after passing in features (input) transformed by an activation function and taking the difference between the actual labels and the prediction (output). The ANNs of this work use a greedy algorithm known as backpropagation which provides a fast solution to pattern recognition in this supervised learning experiment. In addition, the ANNs' hidden layers are changed often to represent complex models. More details on the performance of neural networks can be found in [8, 16]. Random forests are comprised of multiple decision trees which are generated binary trees that pivot on the dataset features. Random forests make predictions based on a majority vote of the collection of decision trees. More details of the performance of random forests can be found in [20].

## 2.2 Related Work

While the use of machine learning ensembles is being applied to multiple fields of application, it is important to acknowledge the work of others to acquire new perspectives, to prevent duplication of another work, and explore new approaches. This section is a summary of the works reviewed relating to this thesis.

### 2.2.1 Active OS Identification

In [21], five algorithms from the popular data mining toolkit Weka are used to generate fingerprints to identify several OSs. These algorithms include J48, JRip, random forests, support

vector machines (SVMs), and IBk. The algorithms are applied to packet responses that were obtained from probing the hosts on the network. This active approach uses carefully constructed TCP/IP packets designed to strengthen implementation differences between OSs. The features of this work used multiple fields from the TCP/IP stack of IPv4 packets. It is mentioned that the tool only used crafted IPv4 packets so the dataset consists of only IPv4 packets. The work could be applied using IPv6 packets as the TCP/IP fields should mostly match, but this exploration was not mentioned. This work is based on [6] where conjunctions and decision lists were used to generate fingerprints. The exploration of using IPv6 packets is a central goal of this thesis that differed from this work in OS fingerprint generation.

Another work [22] uses neural networks for operating system detection between Windows versions or between several OSs. When distinguishing between Windows versions, a 3-layer supervised learning neural network uses 413 features, 42 neurons for the hidden layer, and 25 outputs. Remote Procedure Call (RPC) query response packets contain unique identifiers that correspond to details of programs installed on the remote host. These identifiers are used as the 413 features, while each version and edition of Windows and service pack of Windows correspond to each of the 25 outputs. While the neural network is similar to a neural network setup in this thesis, our neural network has considerably less neurons across all layers. Training of this neural network takes 14 hours, while the training of our neural network ensemble takes several seconds due to the smaller size of each layer. In addition to using neural networks, this work continues to use a combination of signature databases and conditioned analysis to further refine their system, unlike the work in this thesis that does not use any databases.

Another active approach uses Nmap to fingerprint the OS of IPv6 packets [11]. There are 154 crafted network packets that are used to probe the hosts on the network. These hosts will

respond and those response packets will be analyzed for features. Their dataset consisted of 26 different OSs and versions. Similar to common fingerprinting techniques, the work analyzes several TCP fields for OS identification, but also includes the use of ICMPv6 and NS packets to determine response implementation differences when probing the hosts. The IPv6 features used are total packet length, value of the traffic class, and guessed initial hop limit. The machine learning technique used in this work is a L2-regularized logistic regression linear model. This work decided to use the traffic class as they are also trying to identify OSs remotely, while this thesis does not use the traffic class as our dataset is reliant on the local area network SLAAC process which does not require the traffic class for any component of the communication. Additionally, 154 network packets probing each host can be notable, which is the drawback to active approaches. Furthermore, the work decided to use a linear model, but arguably the number of fields used and data are too complex to conveniently fit into a linear model. They were able to achieve 70.2% accuracy, but this could have been improved by using a more complex model that can better fit to the data such as neural networks or random forests.

In [9], IPv4 OS fingerprinting methods are applied to IPv6 along with newly enabled methods to confirm that fingerprinting methods did not fundamentally change. Some methods that were common for both protocols were identification through fragmentation flags, TTL or hop limit, and ICMP unreachable port messages. Newly enabled methods mentioned include identification through IPv6 extension headers, MTU discovery, and NDP. Eckstein references another work [4] that NDP has implementation differences between operating systems, but not enough to distinguish OS versions. The work in [4] used several different OSs that are different with the OS versions used in this thesis, and took an active approach to generate fingerprints because NDP based passive IPv6 OS fingerprinting was determined to not be feasible. However,

this thesis shows that NDP based passive IPv6 OS fingerprinting is feasible through the use of machine learning ensembles.

### 2.2.2 Passive OS Identification

Passive tools are constantly evolving in the research field [3] but lack the same level of interest as the amount of effort put into active tools. A passive approach like [5] uses another machine learning algorithm, Naive Bayes classifier, to identify OSs based on TCP/IP fields. This approach used features from IPv4 packets such as Time To Live (TTL), presence of a Don't Fragment (DF) bit, TCP window size, SYN packet size, and TCP options. Moreover, one goal of this work was to determine the number of hosts behind NAT devices, which will be obsolete in IPv6 as NAT will not be used and inapplicable. Various approaches to OS identification, such as this one, use multiple fields of TCP, which is different from this thesis as the number of TCP fields used is kept to a minimum.

The most popular tool used for passive OS identification is p0f [26]. The idea for this tool dates back to June 10, 2000, and is used in a variety of projects that perform passive attacks such as ettercap, Satori, pfsense, and PRADS. Documentation of these projects often point out important IP header fields in the identification process, such as TTL, type of service, and flags set in the IP header. Although some of these fields do transfer over in functionality and name change, such as TTL to Hop Limit, there is less mention of which IPv6 header fields are used in the OS fingerprinting process. Furthermore, p0f has not been worked on in 3 years since 2014, accordingly to its GitHub [27]. Therefore, as new OSs are released, updates to its signature database is required to keep a high OS identification accuracy but has not been apparent on either the host website or GitHub. p0f and other passive tools that depend on signature databases are

reliant on constant development, whereas machine learning ensembles can use the same or similar configurations to determine any additional OSs.

### 2.2.2 Machine Learning in other IPv6 areas

Machine learning techniques have been applied to other IPv6 areas, such as [12]. An unsupervised machine learning DBSCAN algorithm is used for clustering active IPv6 addresses to generate potentially valid addresses. While the IPv6 address space is huge, this approach achieved a 40% success rate of generating valid addresses. The dataset is composed of 3.5 billion IPv6 addresses and no additional features. At first, IPv6 addresses were almost considered as features for this thesis, but the educated outcome from generating IPv6 addresses is that the characteristics of IPv6 addresses can help predict other IPv6 addresses but not necessarily any other characteristics that relate to operating system implementation differences. Their dataset was split into servers, routers, and clients, but all three of these sets can have multiple options for OSs, so it did not seem viable to use an IPv6 address as part of OS identification.

# 3. APPROACH

## 3.1 Methodology

The methodology is to choose and compare machine learning algorithms to identify an operating system (OS) based upon a set of features that can be extracted by passive observation of IPv6 network traffic. The chosen machine learning algorithms for this work are neural networks, neural network ensemble, decision trees, and random forests. Then, IPv6 network traffic is gathered from a computer network that has computers running known OSs. The computer network traffic is analyzed to determine a set of features that are useful for identifying the OS. Next, the machine learning algorithms are trained using a portion of the measured computer network traffic and tested on the remaining portion to determine how effective each algorithm performs. The machine learning algorithms, data collection, feature selection, machine learning training and testing, and accuracy calculation are described below.

### 3.1.1 Neural Network Ensemble

A neural network ensemble consists of multiple neural networks as shown in Figure 3. Multiple neural networks predict the OS based upon features and then a voting scheme or another neural network is used to give the final identification based on the predictions from the neural networks. In this work, the neural network ensemble determines the operating system of a packet by using an internal unanimous voting system. Each neural network is configured diversely with various number of weights. This design allows for the neural network ensemble to encompass more accurate predictions for different types of packets that have varying parameters.

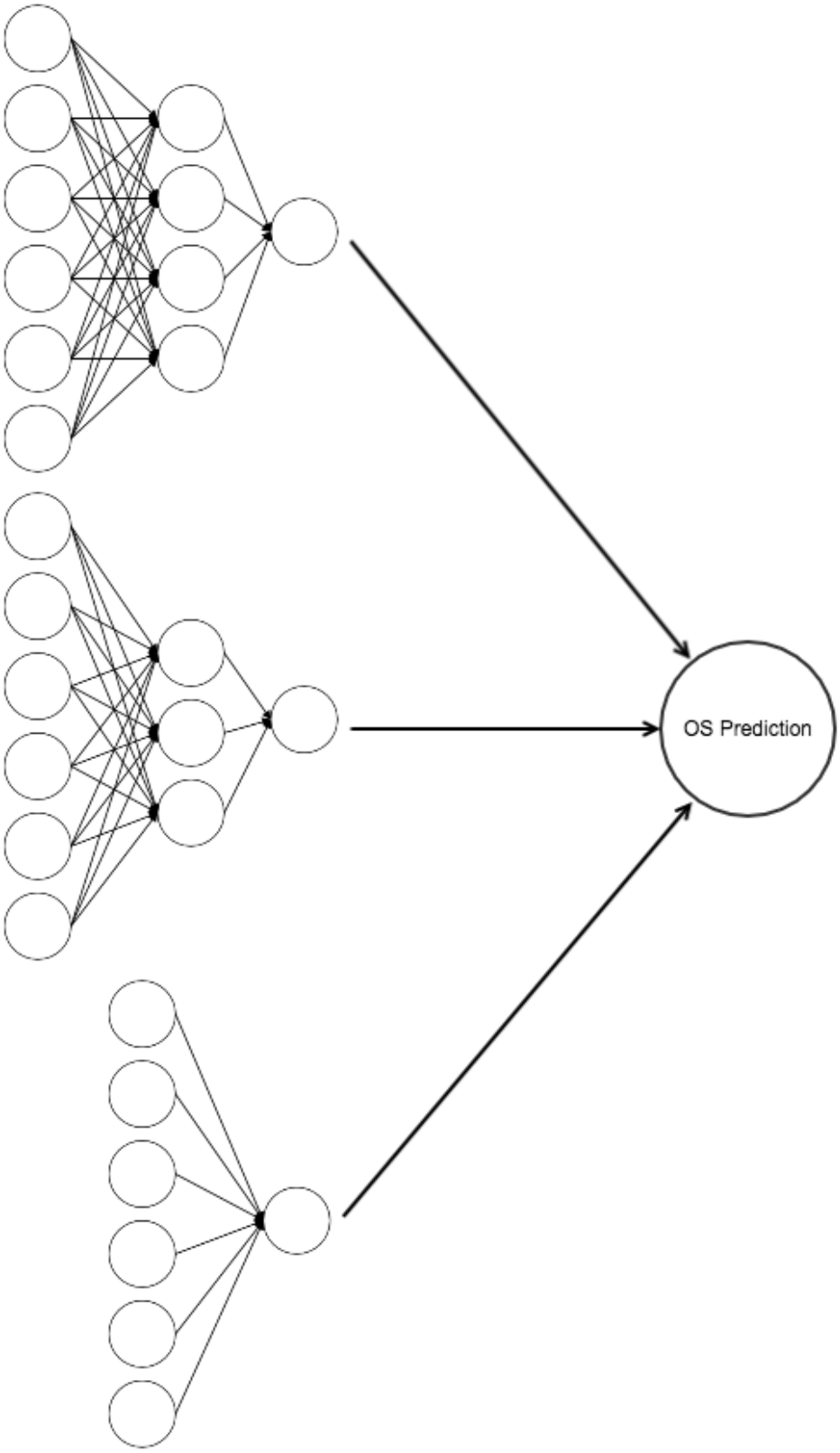


**Figure 3. Neural Network Ensemble**

### 3.1.2 Decision Trees and Random Forests

Random forests make a prediction based on a collection of decision trees that provide individual votes for a prediction. The random forest ensemble identifies the OS based on the prediction with the most votes. Several layouts of random forests are implemented and compared to evaluate the optimal number of decision trees to identify the operating systems. As shown in Figure 4, multiple decision trees give an OS prediction for any particular packet, and the random forest will make a prediction for the ensemble of decision trees based on which OS received the most votes.

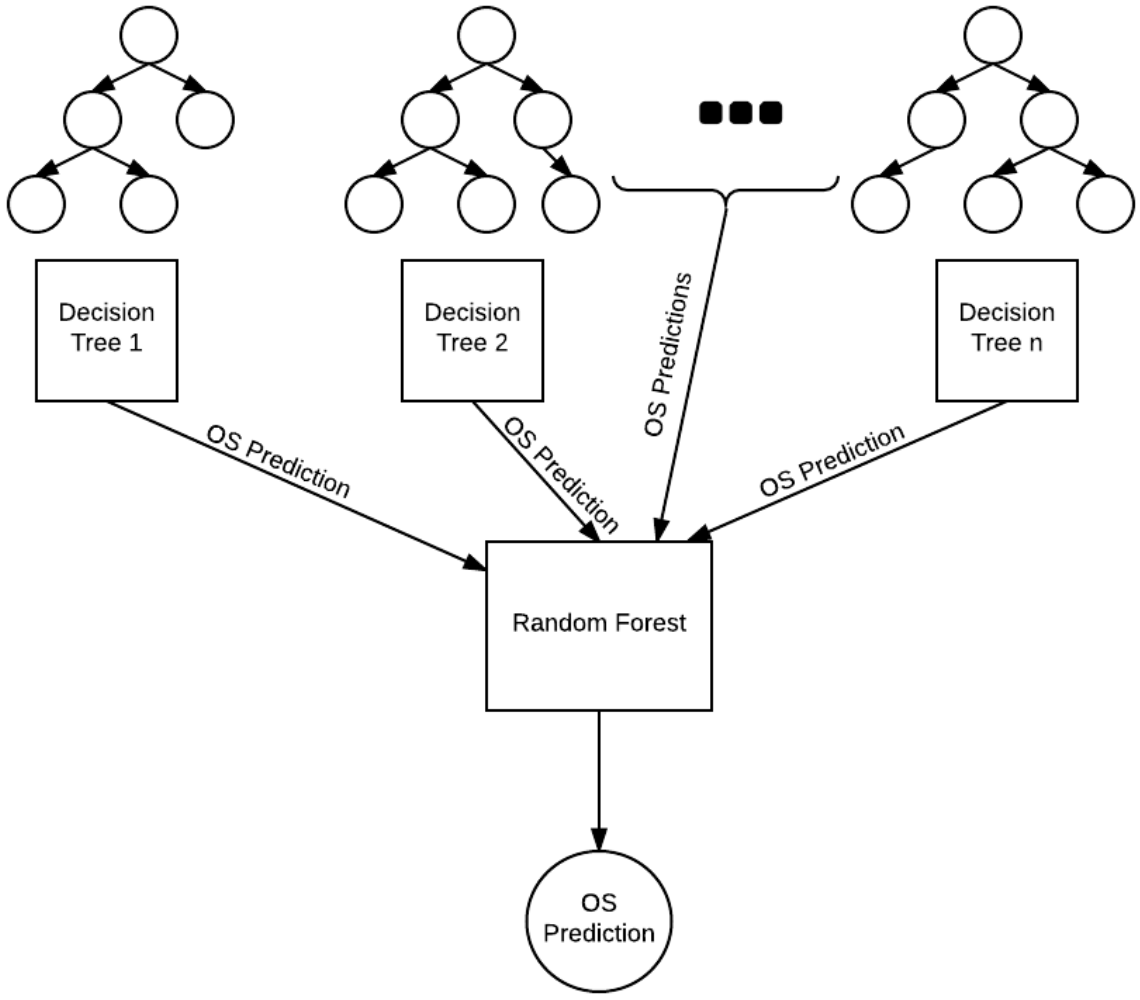


**Figure 4. Random Forest**

### 3.2 Data Collection

Twenty dual-boot computers are used to collect the Linux and Windows IPv6 packets. Wireshark, an open source packet analyzer, is used by a laptop to capture all traffic. The data collection setup is shown in Figure 5. Although all the computers are connected to the same switch, without special configuration on the switch the laptop will only collect a portion of all traffic because NS and NA packets may be sent to a unicast address where only the sender and receiver can see that packet. This is addressed by configuring the switch with a port that has mirroring enabled to send a copy of all traffic to the one port where the laptop is connected, allowing capture of any traffic that flows through the switch. The router in this setup allows for a completion of the NDP and SLAAC between all hosts.

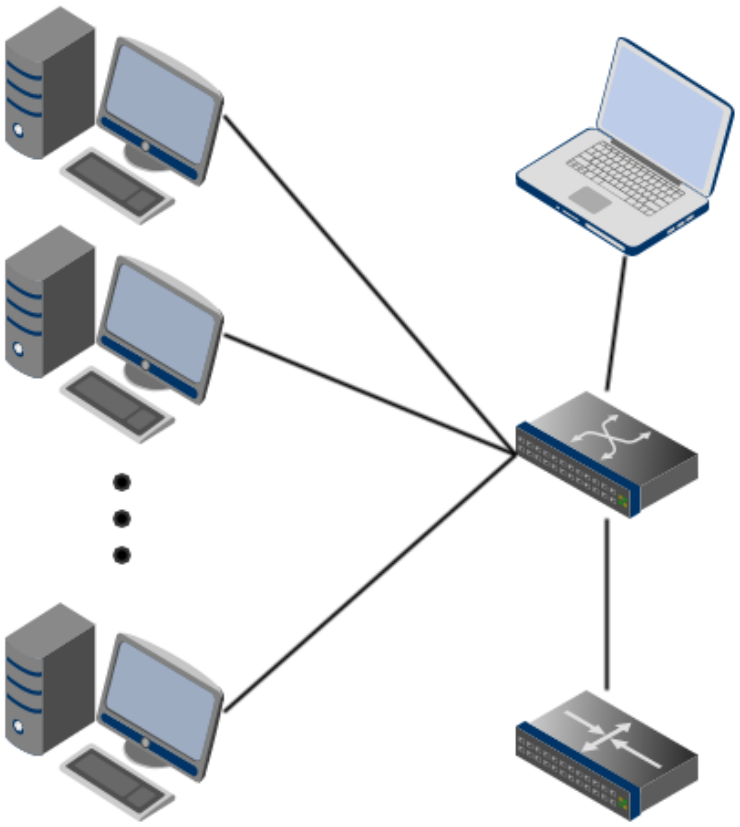

**Figure 5. Data Collection Setup**

To ensure the capture of all packets during the NDP, the router and switch are turned off. There is a total of 50 computers, 20 dual-boot Windows/Linux computers and 30 Mac computers. With so many computers, data is collected separately in sets of 10 computers. As shown in Figures 6 and 7, ten computers are connected to the switch as well as the router and the collecting laptop which is connected by the yellow UTP cable. All computers are booted to run the same OS and are on standby at the login screen to prevent additional services from starting and creating noise from application data. Even without a user login, every machine will participate in the Neighborhood Discovery Protocol (NDP) to obtain an IPv6 address. Once the setup is ready, the laptop begins capturing live network data using Wireshark. Then, both the switch and router are powered on at the same time. The packet collection is flooded with RA, RS, NA, and NS packets until each host, including the router, has assigned a link-local and global IPv6 address to an interface. After a few minutes, the transmission of these packets slows down and then each computer will occasionally send a NS and NA packet to all neighbors to keep an updated list of all neighbors. The setup and process is repeated with all computers booted with each OS, Windows, Linux, and Mac OS. There are three datasets, Windows 10, Linux Ubuntu 16.04 LTS, and Mac OS X El Capitan version 10.11.6.

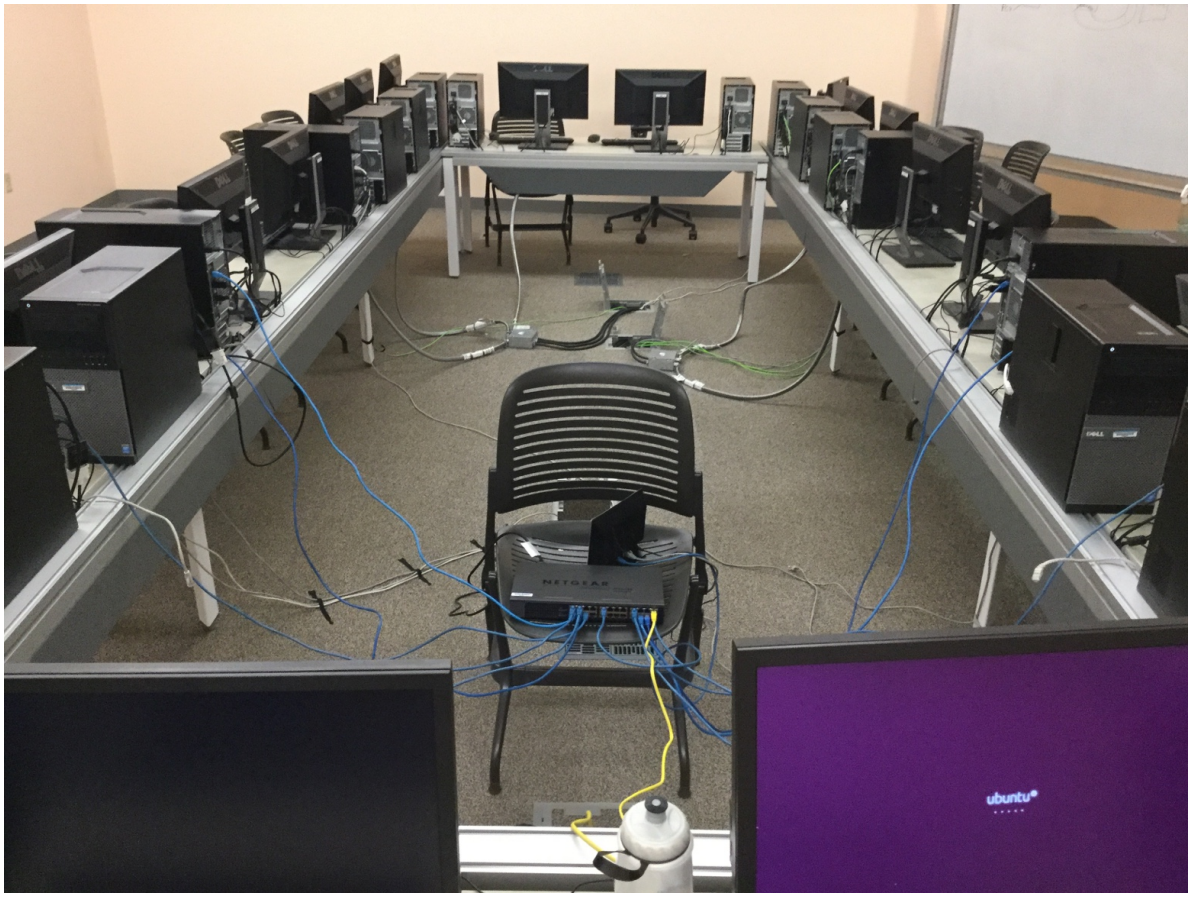


**Figure 6. Data Collection Photo 1**

Once data collection is complete, the pcap (packet capture) files contain more information than needed. Using Wireshark's filtering system, any non-IPv6 packets and any packets that originated from the link-local or global IPv6 address of the router and collecting laptop are removed. Only packets that were sent from the fifty computers remain. To use the packet information outside of Wireshark's interface, the data is exported to a PDML XML file. In this format, the contents of each packet can be filtered using regular expressions. After deciding which features are needed, which is discussed in Section 3.3, a parser program, coded specifically for this experiment, extracts the desired features from the PDML XML file and translates the requested information into an Attribute-Relation File Format (ARFF).

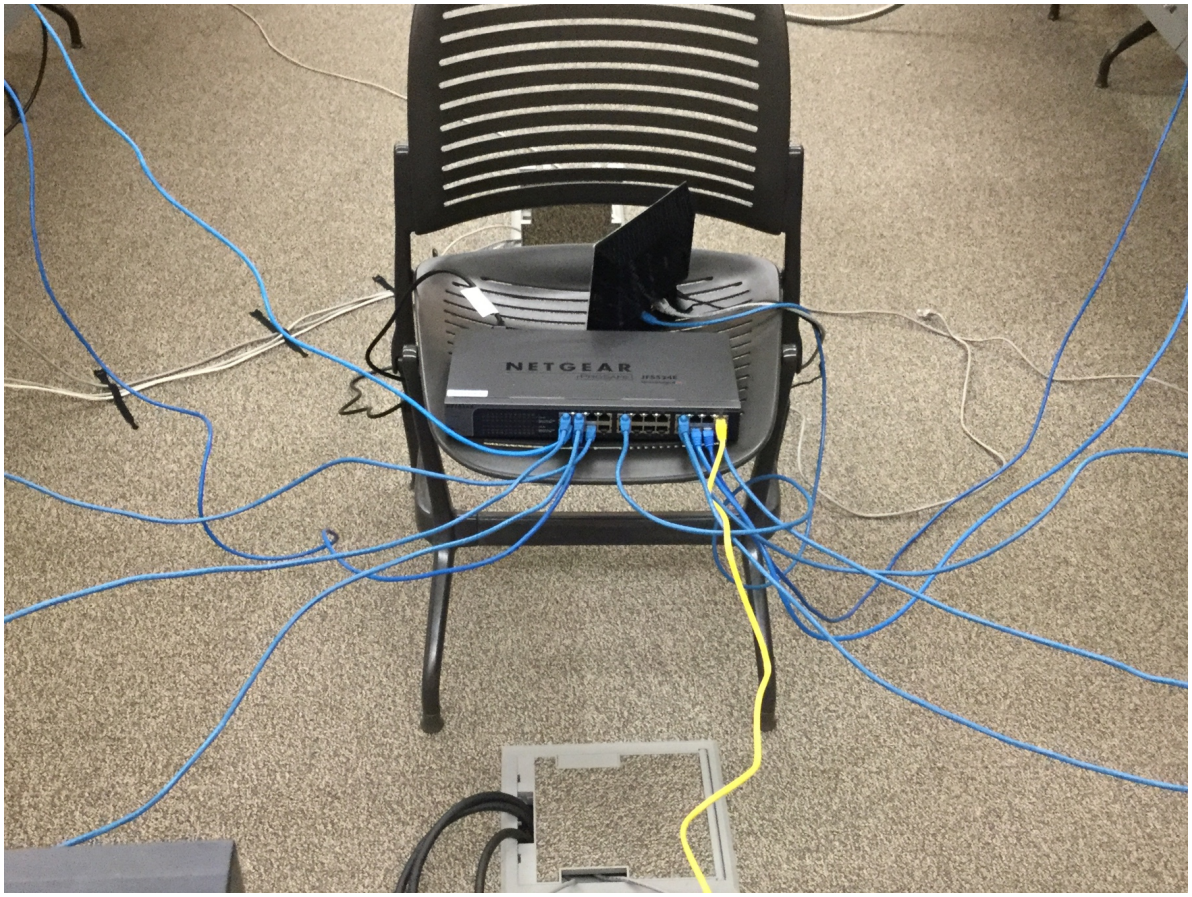


**Figure 7. Data Collection Photo 2**

Table 1 categorizes the number of packets per OS and the number of packets from that OS that were used in training and testing of the machine learning models. Theoretically, more packets from one OS would bias and overfit the model towards that OS. This is not the case for the neural network model. The models require more packets from the Linux OS to reduce the false identification of Linux packets when compared to Windows packets. Even with the inclusion of extra Linux packets, the model does not appear to overfit and favor a Linux prediction. The majority of the packets used are ICMP packets from the NDP, with the remaining packets consisting of DHCPv6 solicitations from NDP, UDP standard queries, DNS queries, and errors.

**Table 1. Number of Packets per OS**

| OS | Overall | Average Training Set | Average Test Set |
|---|---|---|---|
| Windows | 6,482 | 5,186 | 1,296 |
| Linux | 9,494 | 7,595 | 1,899 |
| Mac | 2,193 | 1,754 | 439 |
| Total | 18,169 | 14,535 | 3,634 |

### 3.3 Feature Selection

The IPv6 header is shown in Figure 8. Every header field in the IPv6 header was evaluated to be a possible feature for the machine learning models used for OS identification. As the number of features increases, input nodes for neural networks will increase and decision trees will require more splits. Increasing features can exponentially increase the run time and training time of both models. Therefore, each feature is carefully considered as to whether the feature could potentially aid in the OS identification by IPv6 packets.

Source address and destination address were taken out as features as these addresses are created based on either prefixes given by routers, derivatives of the MAC address on the network card, or the most likely case of being randomly generated as recommended in RFC 4941 [17]. The version field indicates whether the packet is using IPv4 or IPv6 and this work is only using IPv6 packets for OS identification, so the version field is not necessary as a feature. As described in RFC 6437 [1], the flow label can use a uniform pseudo-random generator to create a flow label value for a given transport session. Randomly generated values will not aid in the identification of OSs, so the flow label field is not used as a feature. The traffic class is a

prioritization feature in IPv6, but the data is comprised of packets from the NDP where no prioritization takes place and the value of this field is always zero, so the traffic class field is not used as a feature. The three remaining fields that are used as features are the payload length field, next header field, and hop limit field.

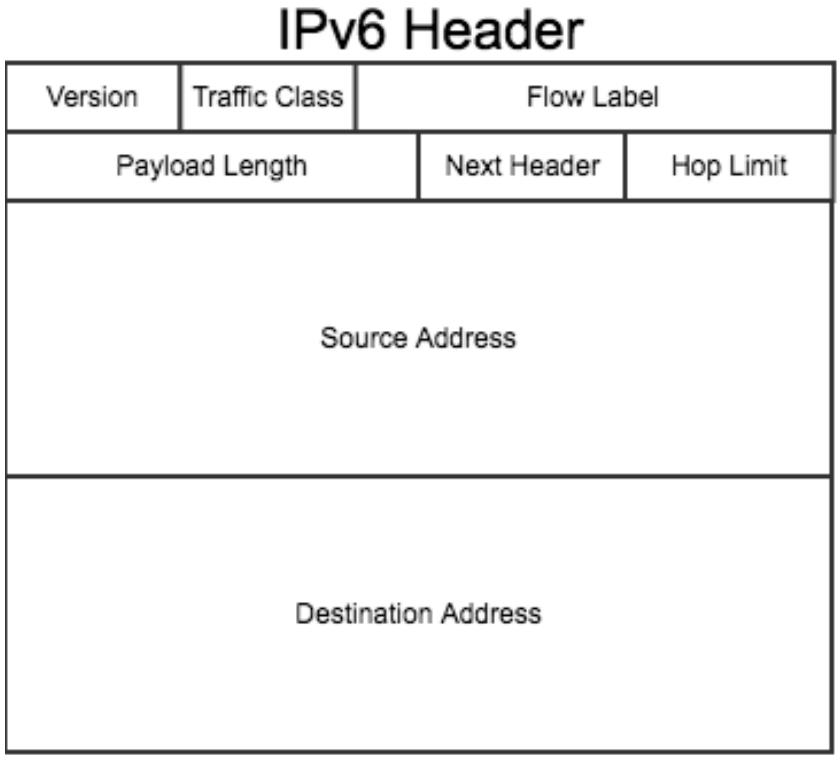


**Figure 8. IPv6 Header**

Three features from the network layer may not be sufficient data for the machine learning models to distinguish OS characteristics. Looking at the next layer of the OSI model, the transport layer, was under consideration, but after inspecting several packets there are many more protocols that had to be taken into consideration. With multiple protocols, the fields within these protocols varied and the machine learning models would require an input from the field whether the field existed or not for any particular packet. If the field did not exist, then the question was what to put as input for that field. Inputting a "0" for a non-existent field is not

acceptable considering that a zero value still has value and can misrepresent the data. With all these considerations, the next option was to explore the metadata of the packet, the frame pseudo-protocol, which is generated from Wireshark's pseudo dissector that does not show fields that actually appear from the packet, but are relevant to the packet [19]. Upon manual inspection of the metadata, there were a few fields that were subtlety different. These fields include the packet size, the protocols used, and the transport layer used for a given packet. These metadata fields were always present since the frame protocol is at the top of every protocol tree. Knowing the contents of the transport layer may prove most useful, but just knowing the transport layer protocol used for a given packet was sufficient. The final set of features used to do OS identification are listed in Table 2 below.

The packet size is the size of the network packet in bytes. The protocols feature is a string that lists all the protocols from the physical layer up to the application layer that are used in the packet. The transport layer protocol feature is a string that shows the transport layer protocol used in the packet, which in this work is mostly ICMP and UDP. The two previous features are considered categorical values and are enumerated before being used in the machine learning ensembles. The IPv6 payload length is the length of the content in bytes, this does not include the size of the IPv6 header. The IPv6 next header is an optional extension header that allows for additional IPv6 packet options. The IPv6 hop limit is a counter that is decremented at every intermediate node before the destination. The packet is discarded when this value reaches 0. Some protocols require this to be a specific value for certain messages to be valid, like the maximum value at 255.

**Table 2. Feature Set**

| Protocol | Fields |
|---|---|
| Frame (packet metadata) | Packet Size (Numerical) |
| | Protocols within Packet (Categorical) |
| | Transport Layer Protocol (Categorical) |
| Internet Protocol version 6 (IPv6) | Payload Length (Numerical) |
| | Next Header (Numerical) |
| | Hop Limit (Numerical) |

### 3.4 Neural Network System

The Java neural network toolkit used in this work is based on the open-source machine learning toolkit Waffles [13] to build an OS identifier. The goal of this approach is to use a neural network to perform passive OS identification. A neural network with a feed-forward backpropagation gradient descent using no hidden layers was first used to establish a model to perform OS identification with a small data set of roughly 2,000 packets containing only IPv6 packets from the Windows OS and Linux OS. Experimentation on the learning rates, momentum, activation functions, and additional hidden layers with varying sizes were used to determine the best configuration for learning IPv6 packets. Figure 9 presents a neural network setup with one hidden layer using three neurons. With a learning rate that is too small, the convergence may be very slow, while a high learning rate may cause the model to never converge. The preferred learning rate was found to be 0.01 to balance these two issues. Momentum has a noticeable difference in accuracy. Through several runs using a momentum

ranging from low momentum to high momentum it was empirically determined that a good momentum lies at 0.5. Several activation functions were used such as the hyperbolic tangent function “tanh”, identity function, inverse tangent function “arctan”, logistic function, and rectified linear unit. The tanh function was proven best with accuracy of OS identification, while the other activation functions gave poorer results in accuracy.

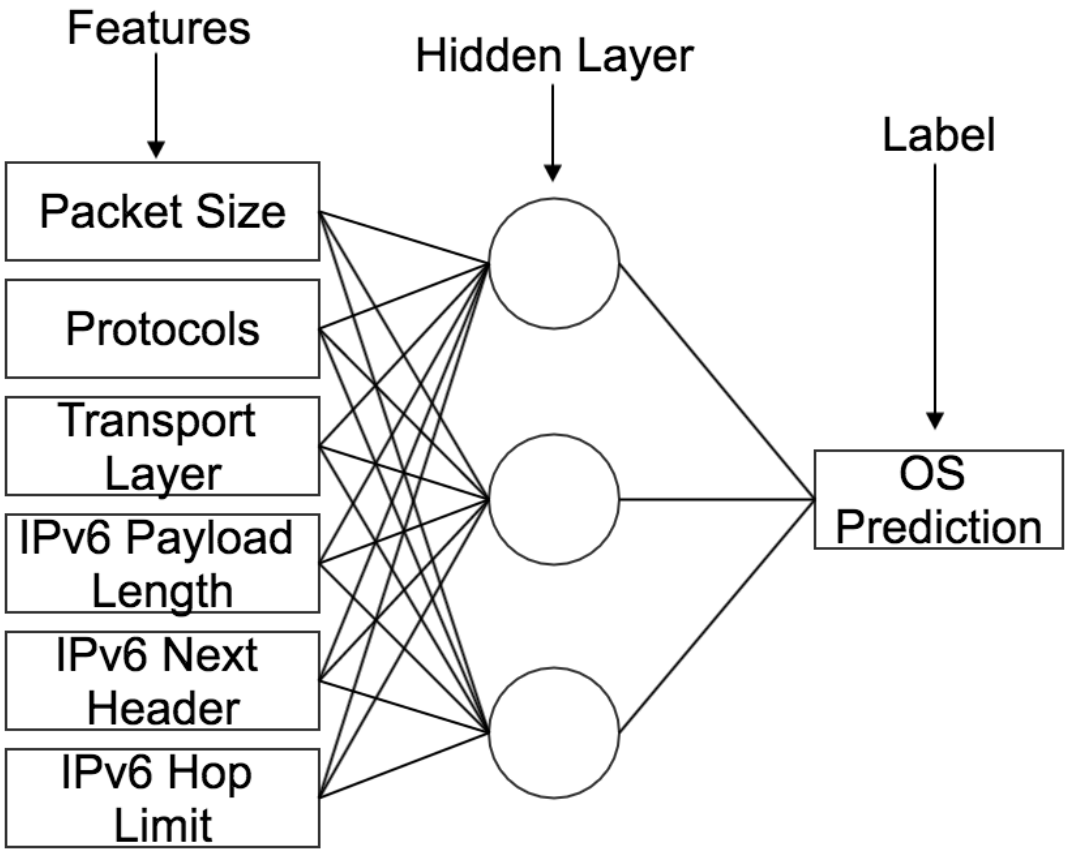


**Figure 9. Neural Network Layout**

After fine tuning the parameters, the average accuracy of the neural network was approximately 65% which is not satisfactory. Adding a hidden layer of any number of units gave a slight decrease in average accuracy. In order to determine how to improve this model, the gradient is analyzed as the model trains. Through gradient analysis, the model showed that

convergence does take place whether the model had various hidden layers or no hidden layer, yet the accuracy of each setup was within a small range of one another. One interpretation for the neural networks' behavior is that the data has local optima adjacent to the global optimum and the various hidden layer layouts are converging close to one another but are situated in separate local optimum. So, the next step is to see the accuracy of identification per OS. Through different hidden layer layouts, the accuracy of identifying Windows OS was consistent while the identification of Linux OS varied.

A comparison using different layouts through training and predicting OSs show that the hidden layer layout can accurately predict a subset of IPv6 packets that are sent from a Linux OS. Bootstrap aggregation, also known as bagging, is an ensemble method used to take a collective response from several models. Multiple neural networks with differing hidden layers can be used together to represent a more complex model. Therefore, three neural networks of various hidden layouts were created and trained to give individual predictions to be aggregated into a unified prediction. The idea is that whichever OS got the most votes, then that should be the prediction. However, this did not work well. The accuracy did increase but there still was a significant amount of error.

After careful observation between the predictions and actual labels, the label for any given packet was always Linux if any one neural network made a Linux prediction. In other words, the neural networks can individually identify a Linux OS packet but may not be the consensus among all the neural networks. The bootstrap aggregation turned into a Windows OS unanimous voting system, where if any one neural network voted for a Linux OS prediction, then the ensemble will predict Linux and that means Windows OS will require a unanimous vote

across all the neural networks. This new approach proved to be the most effective across multiple runs of the experiment.

The average accuracy of this setup was significantly higher, but the accuracy variance was still high. This is most likely due to the initial weights of the neural networks. When neural networks are created, the weights are set to small random values calculated using the Gaussian distribution. However, this randomness could set the neural network weights close to a local optimum that incorrectly identifies the packets. Experimentation of the initial weight value range showed minimal improvement. Neural network models falling into local optimum does not always provide the best possible solution and is prone to occur for any given dataset. The only combatant to the randomness of weight initialization that was discovered is extra training.

Extra training is the re-training of a neural network if the performance of the neural network is poor. When training a neural network, the features are loaded into matrices, randomly re-ordered and then split into a training set and a testing set. The training set consists of about 80% of the data and allows for the neural networks to learn and adjust weights to better accommodate the data for more accurate predictions. The testing set consists of the remaining 20% of the data and are packets that the neural network did not encounter during training. The testing set is used as a validation on whether the neural network can make accurate predictions. The predictions during validation do not modify or change any weights of the neural network and are compared to the testing set labels. The prediction and label comparison provide a count of misclassifications for the testing set. The number of accurately identified packets are divided by the total number of packets in the testing set to receive an OS identification accuracy for that neural network ensemble. If the neural network does not perform well, extra training reuses the training set to train the neural network again in hopes to leave the local optimum and shift closer

to the global optimum. In the extra training process, the existing neural network is trained on the original training set. Extra training does improve the neural network sometimes. Other times the neural network stay the same, whether it stays in the same local optimum or moved to a different local optimum is difficult to say. Occasionally the extra training may push the neural network into a worse local optimum, which causes more misclassifications. The extra training is a successful technique that improves average accuracy, but does require more time for the additional training.

After identifying a good neural network model, a new dataset is used consisting of Windows, Linux, and Mac. Although the number of Mac machines are equivalent to the number Linux and Windows machines used during data collection, there was significantly less IPv6 packets that came from Macs. This lead to concerns on whether the amount of data from Mac computers was adequate. Without access to more Mac computers, several models were attempted to determine if the neural network system would require a different setup when identifying three OSs.

First, a single neural network is used, but as observed before with the Linux and Windows dataset, the accuracy was poor. Then three neural networks are used and bagging is re-introduced into the voting system. This quickly became an issue on how voting ties should be handled. With three possible predictions, no combination of neural networks could handle ties properly for bagging. For instance, three neural networks can result in a three-way tie, four neural networks can result in a two-way tie of two votes each, five neural networks can result in a two-way tie of two votes each with the last vote going to the third OS, and so forth.

Bagging is only one of the ensemble neural network meta-algorithms. The other ensemble setups that were tested were gating and stacking. Gating, as shown in Figure 10, uses a

neural network interpreter and learns from the predictions of the neural networks to determine what the actual prediction should be by comparing the given predictions to the label. The neural network will learn from the mistakes and successes of the other neural networks. The gating ensemble seems like the neural network interpreter should work well. Yet, gating performed poorly with an average accuracy of 60.8%. The next attempt was to include an additional neural network to the gating ensemble. The interpreter may improve its accuracy with four neural networks. However, the additional neural network worsened the average accuracy to a staggering 46.2%.

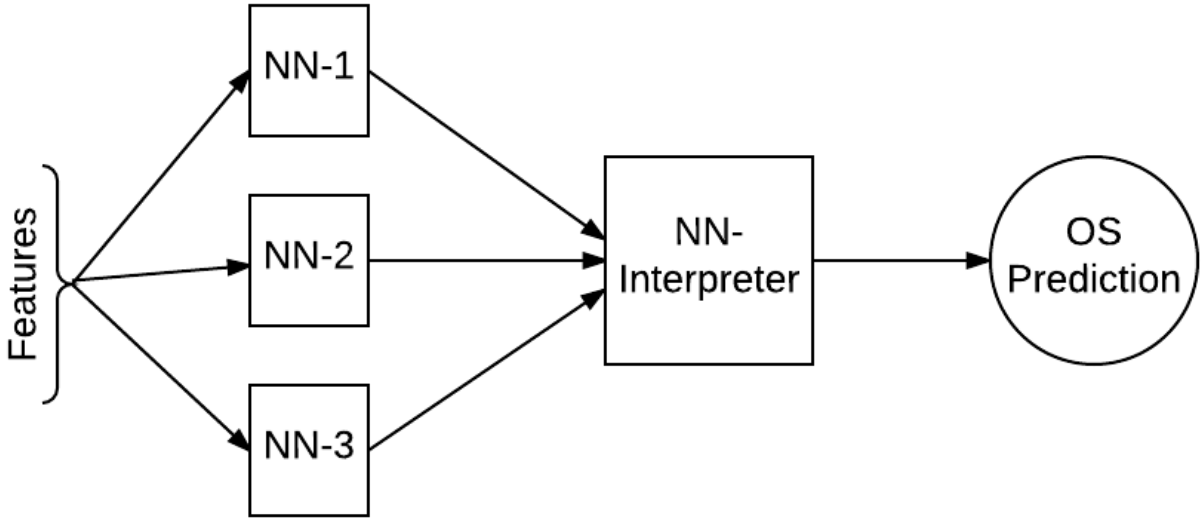


**Figure 10. Gating Ensemble**

With no success using the gating ensemble, the stacking ensemble, as shown in Figure 11, could show promise. The experimentation starts out similarly by using three neural networks to train using only the training set. Then the outputs from these neural works as well as the six features from the IPv6 packet are fed into the neural network interpreter. The result of this setup is an average accuracy of 57.2% with a very similar median and standard deviation to that of the gating ensemble. As tested before, an additional neural network is introduced to see if there

would be any improvements. The ensemble interpreter using four neural networks and IPv6 packet features show an average accuracy of 58.7%. While there was a slight improvement, the standard deviation was still high at about 23.4%. Both the gating and stacking ensembles did not prove to be effective for identifying OSs by the IPv6 packets.

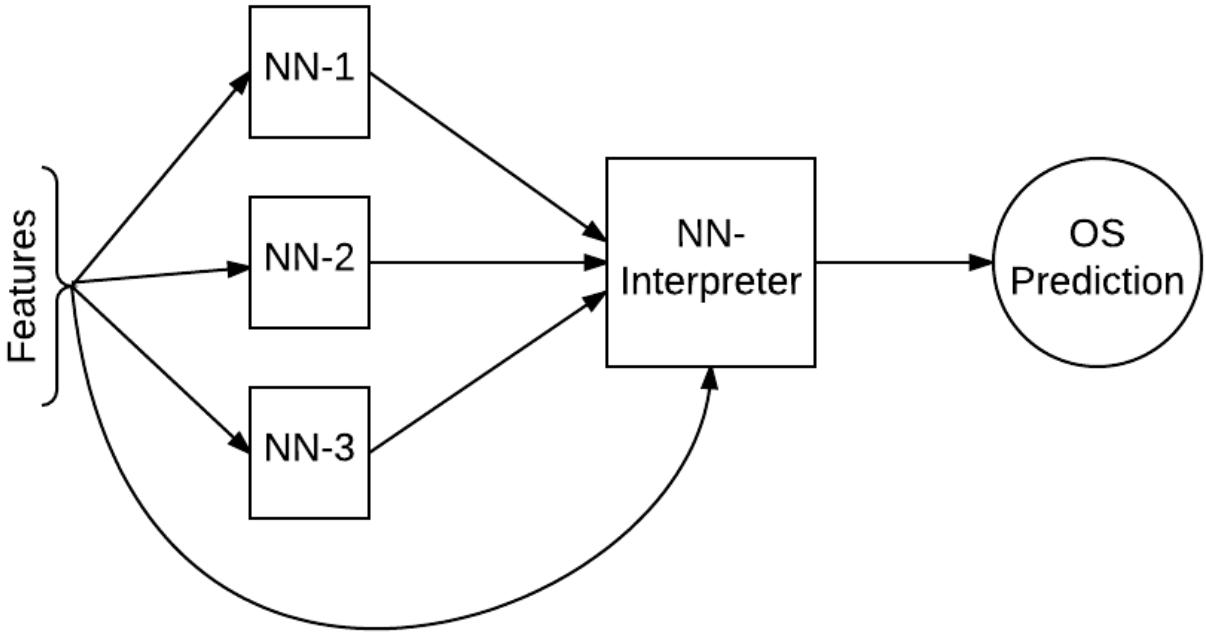


**Figure 11. Stacking Ensemble**

The next configuration explored used two additional neural networks that were trained to only two of the three OSs. One neural network trained on an equal number of packets from Windows and Mac OS, while the other neural network trained on an equal number of packets from Linux and Mac OS. The assumption was that these neural networks would become experts in distinguishing between the two OSs it knew. The two additional neural network datasets are different, but the packets are derived from the original training set.

As shown in Figure 12, the system starts where the first 3 neural networks each make a prediction. If any prediction was for Mac OS, then the ensemble skips the next two neural

networks and votes for Mac OS. Otherwise, if there was a Linux prediction, then the vote will be Linux, unless if either of the two additional neural networks say it was a Mac OS, then the vote would change to Mac OS. Finally, if all three original predictions were Windows OS, then we look at the prediction from the neural network that trained on only Windows and Mac OS packets. If the prediction is for Mac OS, then the vote will be for Mac OS. Otherwise, the same is true for a Windows OS prediction. This ensemble is the best one discovered during experimentation, but with only achieved an average accuracy of 76%.

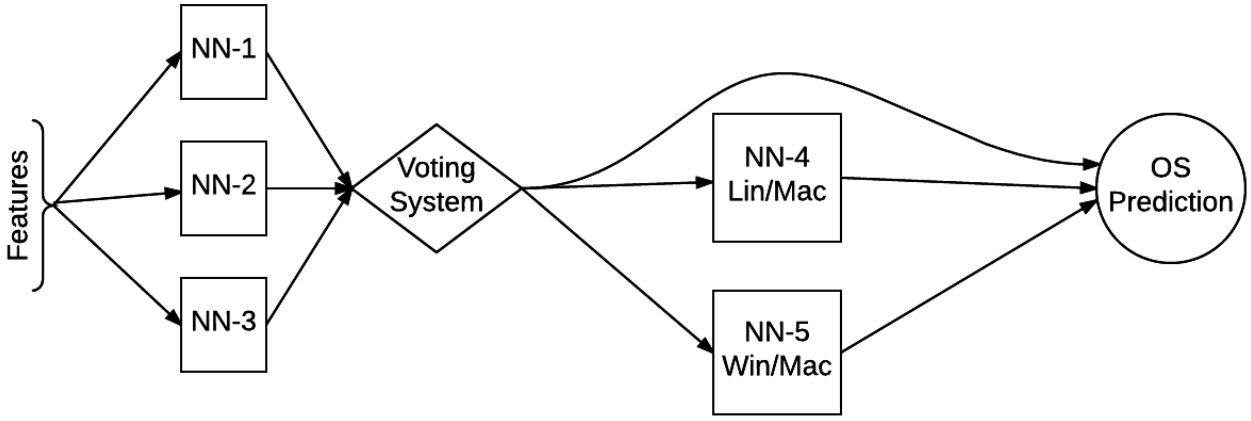


**Figure 12. 5-Neural Network Ensemble for Windows/Linux/Mac**

The last technique introduced into the neural network system is a rule-based pipeline to quickly identify an OS. If any packet contains a particular feature that has only appeared in a single OS, then a rule can be used to forgo all the neural network prediction computations and output the appropriate OS. One rule is implemented in this neural network system to identify a Windows OS if the protocols feature included the Link-Local Multicast Name Resolution (LLMNR) protocol. All the packets were checked for the LLMNR protocol and only appeared for the Windows OS. The LLMNR protocol has been a part of the Windows OS network traffic from the most current version Windows 10 back to Windows Vista. This protocol is cross-checked to see if it could potentially appear in a different OS. There is a daemon known as

LLMNRD for Linux that implements the LLMNR protocol to respond to name resolution queries sent by a Windows client. However, this daemon is not installed in Linux by default and must be installed separately as an additional package if desired. There is potential for more rules for features that are OS exclusive and additional rules can further increase OS identification accuracy beyond neural network predictions.

### 3.5 Random Forests

Random forests are a quicker OS identifier than neural networks. Random forests can consist of *n* decision trees. An increasing number of decision trees will improve OS identification up to a threshold that is dependent on the dataset. Beyond this unexplored threshold, the random forest may not or may insignificantly improve OS identification accuracy. One goal is to determine the appropriate number of decision trees to balance run time and identification accuracy. Several configurations are compared to determine the best number of decision trees without sacrificing accuracy.

The implementation of this machine learning ensemble is simpler than that of neural networks. At first only Windows and Linux packets were used to make sure that the creation and implementation of decision trees are working correctly. The random forest code is based on the open-source machine learning toolkit Waffles [13]. The random forest ensemble was very easy to use and maintained high accuracy after including the Mac OS packets into the dataset. The same datasets from the neural network ensemble were used, as well as randomization and splitting the data 80%/20% for the training set and testing set, respectively.

After ensuring the ensemble was working correctly, the number of decision trees was varied and the average accuracy of 100 experiments was measured. As shown in Table 3, the use of five decision trees seemed sufficient for the dataset. However, thirty decision trees were

used to prevent the concern of the unlikely chance where two or three badly constructed decision trees can negatively influence a five or even fifteen tree ensembles.

**Table 3. Accuracy of Different Number of Decision Trees for a Random Forest**

| Number of Decision Trees | Average Accuracy of 100 experiments |
|---|---|
| 5 | 89.2% |
| 15 | 89.5% |
| 30 | 89.6% |
| 60 | 89.2% |

In general, the use of random forests was more accurate than the neural network ensemble when identifying the OS when the dataset contained Windows, Linux, and Mac OSs. Each decision tree takes roughly less than 25 milliseconds to create so even with thirty decision trees, random forests were much quicker than the neural network ensemble that took 3.4 seconds on average to create and train.

### 3.6 Training and Testing

After training the neural networks on 80% of the measured dataset, packets from the remaining 20% of the dataset are fed through the ensemble to obtain a prediction. The prediction is compared to the withheld label, and if the two do not match then a misclassification count is incremented. Similarly, the random forest is randomly created by training it with 80% of the measured dataset. Then, packets from the remaining 20% of the dataset are fed through the ensemble to obtain a prediction. The majority prediction is compared to the withheld label, and

if the two do not match then the misclassification count is incremented. The reciprocal of misclassifications is the measurement of accuracy for both ensembles.

### 3.7 Accuracy

As the testing set is created, a tally of the number of packets that came from each OS is kept. Since the data is randomized before the training set and testing set split, the number of testing set packets from any particular OS may not be exactly the same between experiments. As misclassifications occur, a separate tally is kept for each OS misclassification alongside an overall misclassification count. The experiment is executed over a hundred times and the misclassifications for each experiment is averaged and several statistics are computed, such as best, median, worst, and standard deviation accuracy. Ensemble accuracy can be computed as:

$$Accuracy\ (\%) = 1 - \frac{Misclassifications}{Number\ of\ Testing\ Set\ Packets}$$

Accuracy for a specific OS is computed as:

$$OS\ Accuracy\ (\%) = 1 - \frac{OS\ Miscalssifications}{Number\ of\ Packets\ from\ OS\ in\ Testing\ Set}$$

# 4. RESULTS AND ANALYSIS

The results of this work consisted of two machine learning ensembles with multiple configurations being applied to two datasets. First, we use a Windows and Linux dataset to a 3-neural network ensemble. Followed by results of the same ensemble with extra training. Then, results from a dataset containing Windows, Linux, and Mac packets used in a 5-neural network ensemble. Next, results from random forests using 30 decision trees are applied to a Windows and Linux dataset, followed by results from the same ensemble to a Windows, Linux, and Mac dataset. Finally, runtimes of all ensembles are discussed.

## 4.1 Results

As shown in Table 4, the overall accuracy over 100 separate ensembles with the Windows and Linux OS dataset is 85%, which is a 20% improvement over the use of one neural network at 65%. Every time the ensemble is created and trained, the Windows identification is always correct, while the Linux identification has some variance and misclassifications. The Linux identification accuracy median is 90%, which shows that at least 50% of experiments consist of 90% or greater accuracy among both OSs. However, in the worst-case scenario, almost every Linux packet was identified incorrectly standing at a 4% accuracy. This worst-case neural network configuration occurs rarely, which may be caused by poor random weight initialization or unusual distribution of training data during data randomization.

**Table 4. OS Identification Accuracy of Windows/Linux using Neural Network Ensemble**

| Accuracy | Windows Accuracy | Linux Accuracy | Overall Accuracy |
|---|---|---|---|
| Average | 100% | 74% | 85% |
| Best | 100% | 93% | 96% |
| Median | 100% | 90% | 94% |
| Worst | 100% | 4% | 43% |
| Standard Deviation | 0% | 24% | 14% |

Figure 13 is a distribution of the 100 experiments for Windows and Linux neural network ensemble accuracy. While a majority of the experiments consist of a high accuracy of at least 90%, about a quarter of all experiments drop in accuracy to around 62.4% to 71.8%. The several experiments that fell below 53% have little to no accuracy with Linux OS identification. The few outlier experiments that performed poorly cause the average accuracy to drop and skews the distribution of ensemble accuracies.

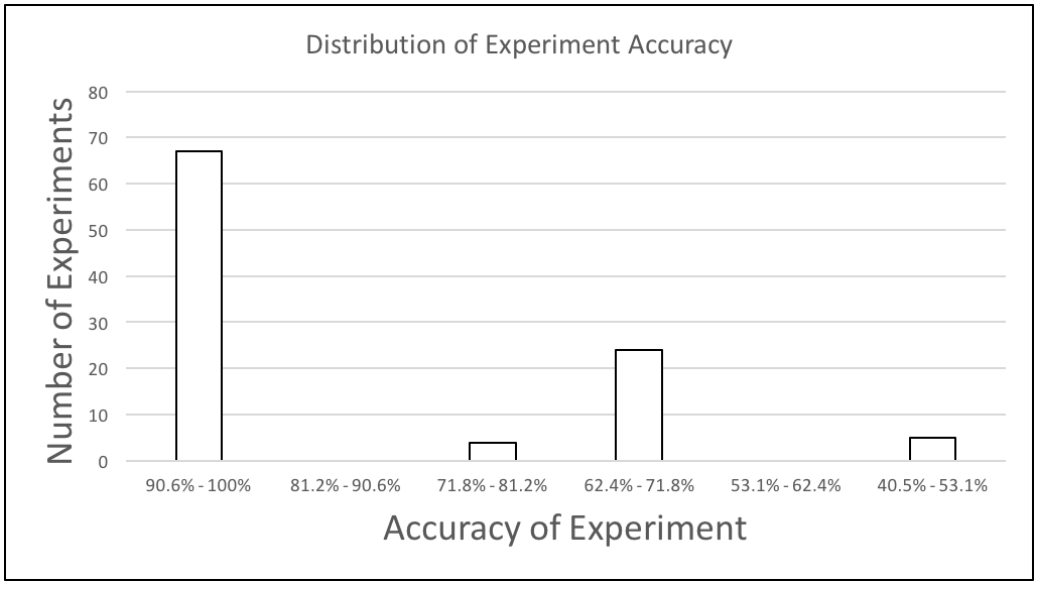


**Figure 13. Distribution of Neural Network Ensemble Accuracy with Windows/Linux**

Table 5 shows the effects of more training on the neural network ensemble. Extra training is given to those ensembles that performed poorly, and reuses the original training set. Extra training has improved the overall accuracy from 85% to 93%. The standard deviation for Linux OS identification is reduced by almost one half, and Windows identification kept its perfect identification accuracy.

**Table 5. OS Identification Accuracy of Windows/Linux using Neural Network Ensemble with Extra Training**

| **Accuracy** | **Windows Accuracy** | **Linux Accuracy** | **Overall Accuracy** |
|---|---|---|---|
| Average | 100% | 88% | 93% |
| Best | 100% | 93% | 96% |
| Median | 100% | 91% | 95% |
| Worst | 100% | 5% | 43% |
| Standard Deviation | 0% | 13% | 8% |

Figure 14 shows a major improvement of the number of neural network ensemble accuracies over 100 experiments. When the ensemble performs unsatisfactorily at less than 90%, using extra training often improves the neural network ensemble resulting with less than ten experiments achieving accuracies less than 90%. Although most of the ensembles were affected positively, there were a few that were affected negatively or no change occurred in accuracy.

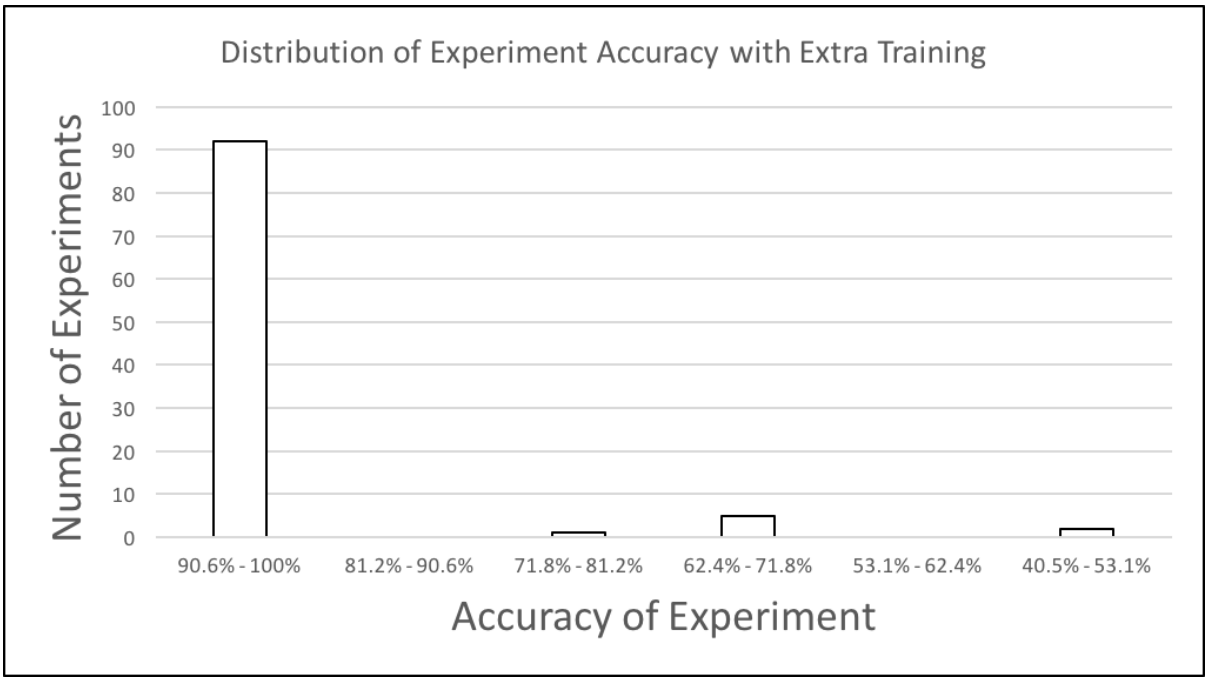


**Figure 14. Distribution of Neural Network Ensemble with Extra Training Accuracy with Windows/Linux**

The results of a 5-neural network ensemble identifying Windows, Linux, and Mac is shown in Table 6. When introducing the Mac OS into the dataset, the Windows accuracy stays preserved while the Linux average accuracy decreases to 75.5%. Linux best and median accuracies are similar to the previous dataset; however standard deviation dramatically increases to 34.1%. In the rare occurrence that the neural networks identified Mac OS packets, the highest accuracy was 75.9%, but most experiments were not able to identify Mac OS features and results in a 0% accuracy. On average, Mac accuracy was low at 6.9% and overall obtained a standard deviation of 20.1%. In the overall worst case of 34.4%, the neural networks were only able to identify Windows OS and incorrectly identify Linux and Mac OS.

**Table 6. Five Neural Network Ensemble Accuracy with Windows/Linux/Mac**

| Accuracy | Windows | Linux | Mac | Overall |
|---|---|---|---|---|
| Average | 100% | 75.5% | 6.9% | 76% |
| Best | 100% | 93.2% | 75.9% | 89.1% |
| Median | 100% | 91.3% | 0% | 83.3% |
| Worst | 100% | 0% | 0% | 34.4% |
| Standard Deviation | 0% | 34.1% | 20.1% | 16.4% |

The above experiment accuracy distribution can be found in Figure 15. A majority of experiments received an accuracy in the 81.2% to 90.5% range. However, when this occurs Mac OS accuracy was always very low to none. In the few instances where Mac OS identification accuracy started to increase, Linux OS identification accuracy dropped, so these experiments overall accuracy fell into lower accuracy ranges.

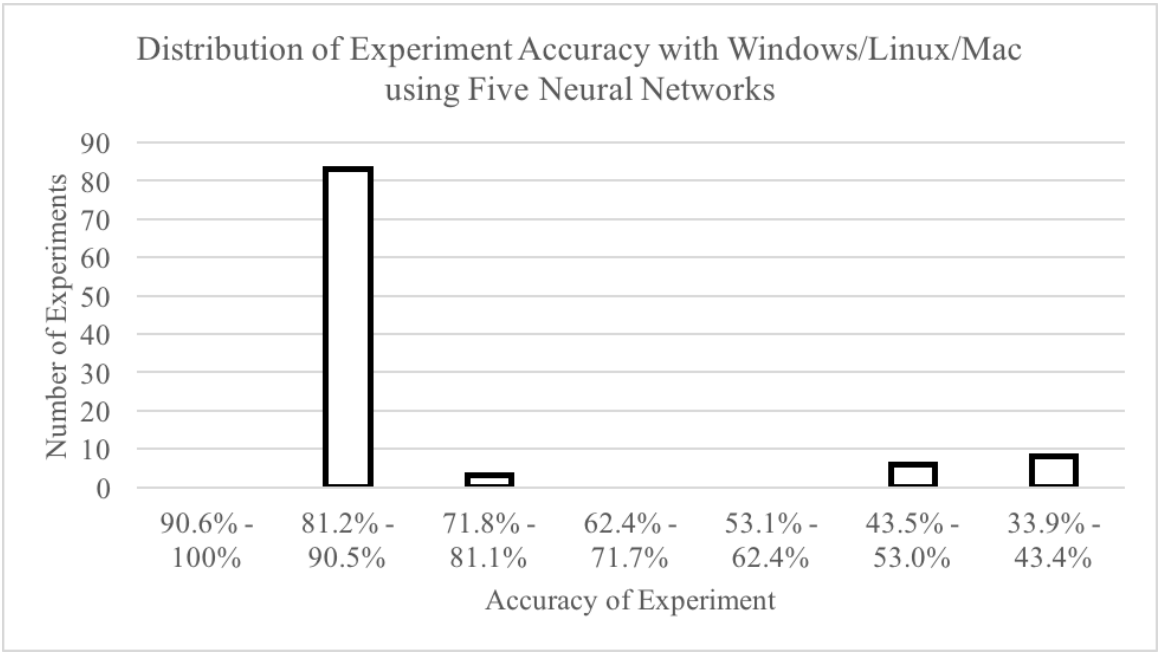


**Figure 15. Distribution of 5-Neural Network Ensemble Accuracy with Windows/Linux/Mac**

Table 7 presents the accuracy of random forests using the same dataset containing Windows and Linux packets from the neural network ensemble. The overall accuracy is 93.6% which is approximately equivalent to the accuracy of the neural network ensemble with extra training. However, using random forests generates misclassifications for both OSs. Furthermore, Windows OS identification is worse with this ensemble as shown by the worst-case accuracy for Windows at 55.5% as compared to the Linux worse case accuracy at 91.7%. This dramatic difference gave Windows a standard deviation of 9.5%, which is greater than six times the standard deviation for Linux at 1.5%.

**Table 7. Random Forest Accuracy with Windows/Linux**

| **Accuracy** | **Windows** | **Linux** | **Overall** |
|---|---|---|---|
| Average | 92.4% | 94.5% | 93.6% |
| Best | 100% | 98.5% | 96.4% |
| Median | 96.9% | 94.5% | 95.3% |
| Worst | 55.5% | 91.7% | 80.7% |
| Standard Deviation | 9.51% | 1.5% | 3.3% |

As shown in Figure 16, the distribution of random forests experiments is better than the neural network ensemble. Any experiment would have at least 81.2% accuracy when using a random forest. Although the number of experiments that achieve at least 90.6% accuracy is not as many as the neural network ensemble with extra training, the random forest distribution does not fall into lower accuracies ranges as does the neural network ensemble.

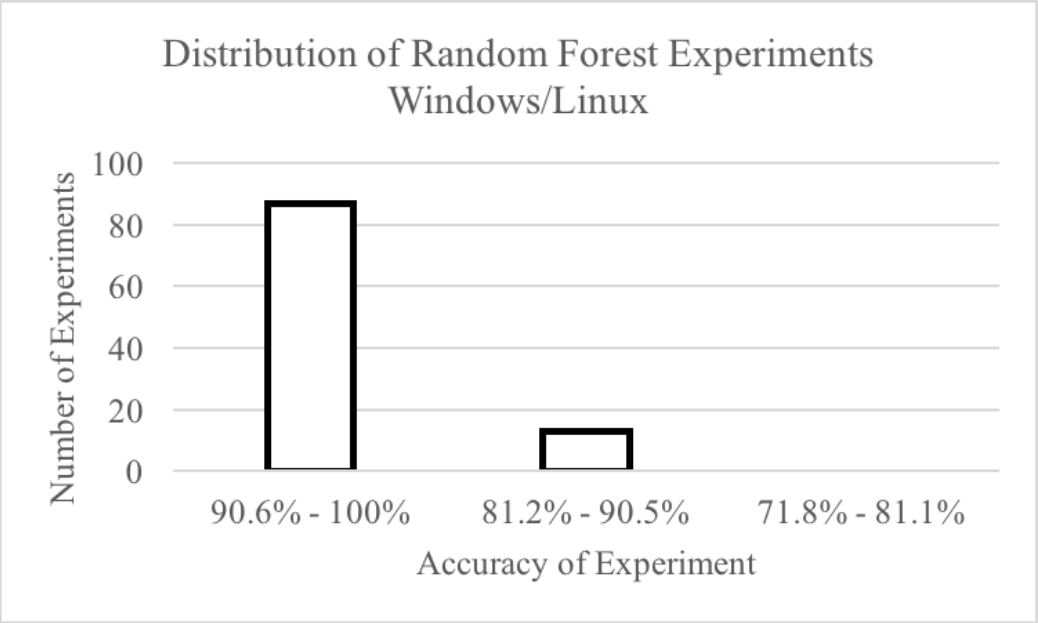


**Figure 16. Distribution of Random Forest Experiment Accuracy with Windows/Linux**

After achieving a functioning high accuracy random forest using Windows and Linux packets, Table 8 displays the statistics of when Mac OS packets are introduced into the dataset. On average the accuracy for this dataset using a random forest was at 89.6%. With this new dataset, Windows OS identification accuracy average is even lower with Mac OS packets in the dataset. The average for Windows average accuracy fell from 92.4% to 88.3%, while still maintaining a best accuracy of 99.9%. Windows still holds the worst accuracy at 54.6% and the highest standard deviation at 10.6% across all three OSs. The average, best, and worst Linux OS accuracies stayed above 90%, which is very comparable when using the previous dataset. While the Mac OS average accuracy lies at 77.2%, the worst-case scenario only fell to 65%, which is much better than Windows worst accuracy at 54.6%. Although the Mac OS identification does not perform as well as the Windows or Linux OS, the variance of identification accuracy is smaller in Mac OS than that of the Windows OS as shown by the standard deviations at 6.8% and 10.6%, respectively.

**Table 8. Random Forest Accuracy with Windows/Linux/Mac**

| Accuracy | Windows | Linux | Mac | Overall |
|---|---|---|---|---|
| Average | 88.3% | 93.4% | 77.2% | 89.6% |
| Best | 99.9% | 97.7% | 95.2% | 93.2% |
| Median | 90.8% | 93.4% | 75.8% | 90.4% |
| Worst | 54.6% | 90.1% | 65.0% | 79.4% |
| Standard Deviation | 10.6% | 1.5% | 6.8% | 3.1% |

As shown in Figure 17, the distribution of experiment accuracy starts to level out the 90.6%-100% range to be roughly equivalent to the 81.2%-90.5% range. There are several random forest accuracies that fell to a lower range of 71.8%-81.1%. These lower accuracies are most likely affected by the poorer performance of identifying the Mac OS packets.

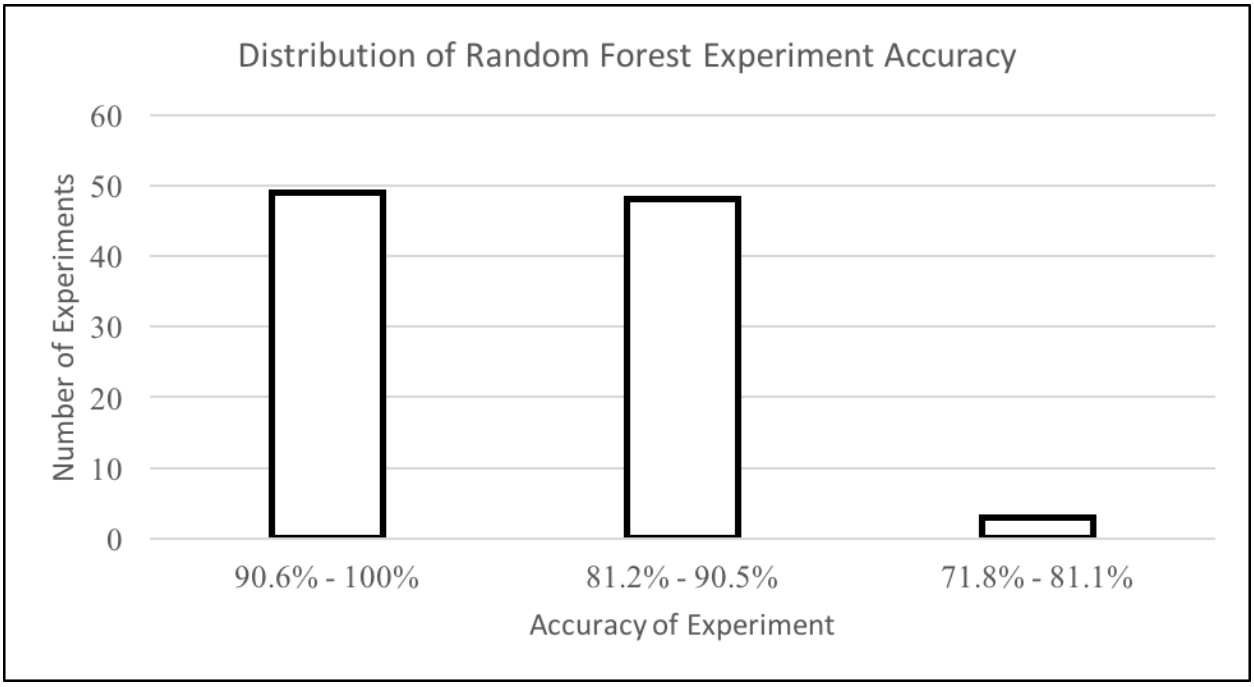


**Figure 17. Distribution of Random Forest Experiment Accuracy with Windows/Linux/Mac**

Table 9 is a list of run times per machine learning ensemble per dataset, which includes both the training and the identification phases. By a wide margin, random forests perform faster than the neural network ensemble counterpart. There is some time deviation between experiments, which is most likely caused by Java's garbage collection running at various instances throughout runtime. Regarding speed, the biggest random forest configuration is faster than the smallest neural network configuration. In general, the ensembles require more runtime as the dataset grows.

**Table 9. Run Times Per Ensemble Per Dataset**

| Dataset | Ensemble | Run Time (100 Experiments) | Deviation |
|---|---|---|---|
| Windows/Linux | Neural Network | 5 minutes 30 seconds | +/- 15 s |
| Windows/Linux | Neural Network with Extra Training | 6 minutes 45 seconds | +/- 17 s |
| Windows/Linux/Mac | Neural Network | 6 minutes 10 seconds | +/- 15 s |
| Windows/Linux | Random Forest 1 DT | 9 seconds | +/- 1 s |
| Windows/Linux | Random Forest 5 DT | 15 seconds | +/- 2 s |
| Windows/Linux | Random Forest 10 DT | 21 seconds | +/- 2 s |
| Windows/Linux | Random Forest 20 DT | 33 seconds | +/- 2 s |
| Windows/Linux | Random Forest 30 DT | 48 seconds | +/- 2 s |
| Windows/Linux | Random Forest 50 DT | 1 minute 10 seconds | +/- 4 s |
| Windows/Linux | Random Forest 75 DT | 1 minute 47 seconds | +/- 6 s |
| Windows/Linux | Random Forest 100 DT | 2 minutes 18 seconds | +/- 10 s |
| Windows/Linux/Mac | Random Forest 5 DT | 19 seconds | +/- 1 s |
| Windows/Linux/Mac | Random Forest 15 DT | 36 seconds | +/- 3 s |
| Windows/Linux/Mac | Random Forest 30 DT | 1 minute 4 seconds | +/- 7 s |
| Windows/Linux/Mac | Random Forest 60 DT | 2 minutes 3 seconds | +/- 14 s |

All ensembles ran on the same hardware using a 1.6 GHz Intel Core i5 processor with 8 GB 1600 MHz DDR3 RAM. Tables 10 and 11 show the run times broken down by the training phase and testing phase. The training phase consists of initializing the ensembles as well as going through the training set. The testing phase consists of the ensemble predicting through all the testing set packets and counting misclassifications. The average runtime for training a neural

network is 3421 milliseconds, which is over 7 times the speed of the average training time for a random forest at 441 milliseconds. However, the average runtime for making predictions with a neural network is 7.6 milliseconds, which is faster than the random forest speed at 10.3 milliseconds. In general, the neural network ensemble will require more time for training, but it can make faster predictions than random forests.

**Table 10. Training Time in Milliseconds (ms) per Ensemble**

| Runtime (ms) | Neural Network Ensemble | Neural Network Ensemble with Extra Training | 5-Neural Network Ensemble | Random Forest 30-Decision Trees; Dataset: Windows/Linux | Random Forest 30-Decision Trees; Dataset: Windows/Linux/Mac |
|---|---|---|---|---|---|
| Average | 3421 | 4917 | 5977 | 441 | 559 |
| Maximum | 4150 | 10719 | 6796 | 883 | 766 |
| Median | 3370 | 3845 | 5889 | 426 | 551 |
| Minimum | 3266 | 3280 | 5604 | 364 | 483 |
| Standard Deviation | 154 | 1838 | 256 | 68 | 43 |

**Table 11. Testing Time in Milliseconds (ms) per Ensemble**

| Runtime (ms) | Neural Network Ensemble | Neural Network Ensemble with Extra Training | 5-Neural Network Ensemble | Random Forest 30-Decision Trees; Dataset: Windows/Linux | Random Forest 30-Decision Trees; Dataset: Windows/Linux/Mac |
|---|---|---|---|---|---|
| Average | 7.6 | 8.1 | 15.6 | 10.3 | 13.6 |
| Maximum | 12 | 13 | 22 | 40 | 27 |
| Median | 7 | 8 | 15 | 10 | 13 |
| Minimum | 6 | 6 | 13 | 8 | 11 |
| Standard Deviation | 0.9 | 1.2 | 1.9 | 3.2 | 2.3 |

The training and testing times were calculated using 102 separate ensembles. However, the first two runs were outliers compared to other runs. Due to these outliers, these two runs were taken out and the remaining 100 runs were used to generate the statistics in Tables 10 and 11. One reason for these outliers could be that the hardware is using branch prediction causing every run after the second run to go faster with predicted pre-loaded instructions.

### 4.2 Analysis

There are advantages and disadvantages to each machine learning ensemble when presented with a dataset that contains both Windows and Linux IPv6 packets. While the neural network ensemble guarantees no misclassifications for Windows OS packets, there is a larger variance when identifying Linux OS packets despite the larger number of Linux packets within the training set. Alternatively, the random forest generally has higher accuracies for both OSs but misclassifications can occur for either OS.

When the neural network ensemble predicts Linux on any particular packet, it can be almost guaranteed that the packet originated from a Linux host. Since the only misclassifications occur from packets that originated from a Linux host, there is a small chance that the packet gets identified as a Windows OS packet. Therefore, when looking at final predictions, every Linux prediction must be from a Linux host while a Windows prediction has a small chance that it originated from a misclassified Linux host.

90% overall accuracy was considered a good experiment with the neural network ensemble so in the event that the experiment achieved less than 90% accuracy the ensemble will complete another training iteration using the original training data. Then the accuracy is measured with the same validation data. With this additional step, the average accuracy for Linux predictions increased to 87% with an improvement of 11% from 76%. Moreover, the

worst case increased slightly from 2% to 4%. The neural network ensemble may get closer to the ideal global optimum but depending on the data in the training set, it may get stuck in a local optimum. The extra training allows the model to move along the error surface, and this extra push can often find the global optimum of this complex model. However, it does rarely occur where the model will get pushed into a worse local optimum and in result cause the neural network ensemble to perform worse than before.

When the Mac OS packets were introduced into the dataset, the neural networks fluctuated in accuracy between Linux and Mac OS. Windows preserved its 100% accuracy rating, which could be that the features from a Windows OS packet is very distinguishable from features of other OSs. In many configurations, Mac did not achieve an accuracy above 0%, and even with the two neural networks, a higher accuracy occurred rarely. The additional two neural networks were supposed to distinguish Mac OS packets, but with no success. There may be features that the Mac OS shares or very similar to features of both Windows and Linux OS.

When the Windows and Linux OS dataset is used in a random forest, the runtime is much quicker than that of the neural network ensemble runtime. The neural network ensemble can guarantee that an IPv6 packet originated from a Linux host, whereas the random forest does not have this quality as misclassifications can occur from either OS. In the perspective of an attacker, a guaranteed OS prediction is more beneficial than a highly accurate prediction. While training does take the most time with neural networks, this step only needs to be performed once.

As seen by run times broken down by each ensemble, training for neural networks is significantly longer than training for random forests. This may be a drawback, but the training phase only needs to run once before making as many predictions as desired. When it comes to making predictions, the neural network ensemble is faster when the dataset consists of Windows

and Linux OSs. However, a random forest can execute faster than the 5-neural network ensemble when using a dataset of Windows, Linux, and Mac OSs. Between both machine learning techniques, the neural network ensemble generally has smaller standard deviations than random forests. As mentioned before, each approach is advantageous whether accuracy or speed is more desirable.

However, when introducing a dataset with Windows, Linux, and Mac OS packets, the random forest predicts with higher accuracy on average. This may be caused by consistencies among field values when leaf nodes are being constructed as well as having that value consistency when using the decision trees for predictions. Random forests are the better ensemble to use when datasets include Windows, Linux, and Mac OS packets as they are faster and highly accurate.

Mac OS generally performed poorer when introduced into the dataset. A number of reasons may include not having enough Mac OS packets which can underfit the model, field values matching with other OSs field values, inconsistent field values between Mac OS packets, and overfitting with Windows and Linux predictions. Having enough data is essential for any machine learning technique. With limited resources and access to Mac OS computers, getting more data was rather difficult. A solution could have been to reuse the same packets twice, but that may bias results. When the training set and testing set is created, the ensembles would only have experience with the training set and has never encountered the testing set, which is great for validation. If there were two copies of each Mac OS packet, there is a likely chance that a duplicate packet would be in both the training set and testing set, which would make results not as notable since the ensemble has seen the packet before.

Generally, datasets are best kept roughly equal to reduce bias and overfitting. However, with the neural network ensemble in this work, an equal number of packets per OS in the dataset lacked sufficient accuracy for Linux packets. An increase to the number of Linux packets improved accuracies, while keeping consistent accuracy of Windows predictions. The issue came with Mac OS packets, since there were so much fewer packets, 2193, the dataset could not shrink without underfitting the models. The gradient of the neural network did not stabilize with such few packets, so increasing the dataset size with the data that was available was the best approach to stabilize the neural network ensemble. The random forest performed well, but potentially could be improved further with more Mac OS data.

# 5. CONCLUSIONS

## 5.1 Summary

In this thesis, machine learning neural network and random forest ensembles are developed and validated as a passive OS identification technique using IPv6 features from the network layer of the OSI model and packet metadata. The neural network ensemble model uses three neural networks with varying hidden layers that feed a prediction into a unique Windows-unanimous voting scheme that is on average 85% accurate with Windows and Linux OS packets and can provide a guaranteed Linux host prediction. Random forests are excellent, fast OS predictors especially for IPv6 packets that originate from Windows, Linux, and Mac OS that is on average 89.6% accurate.

In areas of weak performance, a neural network can rely on the predictions from other neural networks. As each neural network has a unique design layout, this allows for each neural network to be proficient at OS identification of any Windows packet and a subset of Linux packets. Each neural network in the ensemble is trained to identify Linux packet features. Therefore, if any of the neural networks in the ensemble identifies the OS as Linux, then most likely it is the Linux OS. This neural network ensemble configuration was able to achieve a maximum of 96% accuracy when distinguishing between Windows and Linux OSs. The average neural network ensemble accuracy in 100 experiments was 85%. However, when the neural network ensemble is given extra training due to poor performance it will achieve an average accuracy of 93% in 100 experiments. Although the neural networks can provide excellent results for this dataset, the reliability and accuracy of this ensemble diminishes when the Mac OS is introduced into the dataset. In general, the neural network ensemble was unable to identify the Mac OS with usable accuracy.

Random forests provide a roughly equivalent accuracy to the neural network ensemble with extra training when distinguishing differences between the Windows and Linux OSs, 93.6% compared to 93%, but provides better accuracy when distinguishing differences among the Windows, Linux, and Mac OSs, 89.6% compared to 76%. A random forest ensemble with thirty decision trees is a faster solution with runtimes six times faster than neural network ensembles while achieving a maximum accuracy of 93.2%. The average random forest ensemble accuracy was 89.6% when distinguishing among Windows, Linux, and Mac OSs.

## 5.2 Contributions

The contributions of this work are that it uses machine learning algorithms, is passive instead of active, and uses IPv6 features. First, in this work machine learning ensembles can learn to identify the different OSs instead of using a constantly growing database of packet signatures. Or the machine learning ensembles can be used in combination with packet signatures to create a more compact representation that can quickly identify the OS. Secondly, this work uses passive identification instead of active identification that is stealthier and does not introduce more traffic into the network. Finally, this work is the first known work that uses passive identification based on IPv6 features. Passive identification tools today use IPv4 features.

## 5.3 Future Work

While this thesis provides the basic framework for OS identification using machine learning ensembles as a tool, more work is needed in several areas. With some ensembles performing better over others on specific datasets, the combination of several various classifiers can create an ensemble with predictions that reduce the variance of different features of the data. Therefore, the use of a neural network, a random forest, and another classifier such as Naive

Bayes could potentially make a machine learning ensemble with an OS identification accuracy that surpasses the ensembles of this work. Another area of improvement is a wider availability of data including more packets from Mac OS as well as applying these ensembles to different versions of the same OS. While the data was collected from separate machines, the OS image is the same for the Windows computers, the OS image is the same for the Linux computers, and the OS image is the same for the Mac computers. Diversity with machines can give a wider range of outcomes with OS identification. Finally, these ensembles have yet to be applied to operating systems running as virtual machines on a host with a different operating system. The use of a virtual interface may affect the IPv6 packet features that are selected in this work.

## REFERENCES


[1] Amante, S., Carpenter, B., Jiang, S. and Rajahalme, J. (2011) IPv6 Flow Label Specification, RFC 6437.

[2] Azouaoui, O., Kadri, M., and Ouadah, N. (2008) Implementation of a neural-based navigation approach on indoor and outdoor mobile robots, Proceedings of the 5th international conference on Soft computing as transdisciplinary science and technology (CSTST '08), 71-77.

[3] Barnes, J. and Crowley, P. (2013) K-p0f: a high-throughput kernel passive OS fingerprinter, Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems (ANCS '13), IEEE Press, 113-114.

[4] Beck, F., Festor, O., and Chrisment, I. (2007) IPv6 Neighbor Discovery Protocol based OS fingerprinting, [Technical Report] RT-0345, INRIA, pp.27.

[5] Beverly, R. (2004) A Robust Classifier for Passive TCP/IP Fingerprinting, Proceedings of the 5th Passive and Active Measurement Workshop (PAM 2004).

[6] Caballero, J., Venkataraman, S., Poosankam, P., Kang, M. G., Song, D., and Blum, A. (2007) FiG: Automatic fingerprint generation, Proceedings of the 14th Annual Network and Distributed System Security Symposium (NDSS ’07).

[7] Cripps, A. (1996) Using artificial neural nets to predict academic performance, Proceedings of the 1996 ACM symposium on Applied Computing (SAC '96), K. M. George, Janice H. Carroll, Dave Oppenheim, and Jim Hightower (Eds.), 33-37.

[8] Domingos, P. (2012) A few useful things to know about machine learning, Communications of the ACM, Vol. 55, Issue 10, 78-87.

[9] Eckstein, C. (2011) OS fingerprinting with IPv6, SANS Institute InfoSec Reading Room.

[10] Eyal, N., Last, M., and Rubin, E. (2015) Comparison of three classifiers for breast cancer outcome prediction, Proceedings of the 16th International Conference on Engineering Applications of Neural Networks (INNS) (EANN '15), Lazaros Iliadis and Chrisina Jane (Eds.), Article 13, 6 pages.

[11] Fifield, D., Geana, A., MartinGarcia, L., Morbitzer, M., and Tygar, J.D. (2015) Remote Operating System Classification over IPv6, Proceedings of the 8th ACM Workshop on Artificial Intelligence and Security (AISec '15), 57-67.

[12] Foremski, P., Plonka, D., and Berger, A. (2016) Entropy/IP: Uncovering Structure in IPv6 Addresses, Proceedings of the 2016 Internet Measurement Conference (IMC '16), 167-181.

[13] Gashler, M. (2011) Waffles: A machine learning toolkit, Journal of Machine Learning Research, 12:2383–2387, ISSN 1532–4435.

[14] Hepner, G., Logan, T., Ritter, N., and Bryant, N. (1990) Artificial neural network classification using a minimal training set- Comparison to conventional supervised classification, Photogrammetric Engineering and Remote Sensing, 56(4), 469-473.

[15] Hjelmvik, E. (2011) Passive OS Fingerprinting, NETRESEC Blog.

[16] Jones, E.R. (2004) An Introduction to Neural Networks, Visual Numerics, Inc.

[17] Narten, T., Draves, R., and Krishnan, S. (2007) Privacy Extensions for Stateless Address Autoconfiguration in IPv6, RFC 4941.

[18] Ordorica, A. and Thompson, D. (2017) Operating System Fingerprinting using IPv6 Packets and Machine Learning Techniques, Proceedings of the 2017 National Cyber Summit Research Track (NCS '17).

[19] Orebaugh, A., Ramirez, G., and Beale, J. (2007) Wireshark & Ethereal: network protocol analyzer toolkit, Syngress, Page 250.

[20] Random Forests for Beginners, Salford Systems, 2014.

[21] Richardson, D. W., Gribble, S. D., and Kohno, T. (2010) The limits of automatic OS fingerprint generation, Proceedings of the 3rd ACM workshop on Artificial intelligence and security (AISec '10), 24-34.

[22] Sarraute, C. and Burroni, J. (2008) Using neural networks to improve classical operating system fingerprinting techniques, Electronic Journal of SADIO, 8(1):35–47.

[23] Sarter, M. (2014) Comparison of IPv4 and IPv6 headers, Infobidouille.

[24] Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., and LeCun, Y. (2014) OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks, Proceedings of the International Conference on Learning Representations (ICLR '14).

[25] Thomson, S., Narten, T., and Jinmei T. (2007) IPv6 Stateless Address Autoconfiguration, RFC 4862.

[26] Zalewski, M. (2012) p0f v3 (version 3.09b). http://lcamtuf.coredump.cx/p0f3/.

[27] Zalewski, M. (2012) p0f v3 (version 3.09b), GitHub repository. https://github.com/p0f/p0f.