LARGE SCALE MACHINE LEARNING FOR THE DETECTION AND CLASSIFICATION OF MALWARE

by

Sean Kilgallon

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Financial Services Analytics

Summer 2018

LARGE SCALE MACHINE LEARNING FOR THE DETECTION AND CLASSIFICATION OF MALWARE

by

Sean Kilgallon

Approved: _____

Bintong Chen, Ph.D.
Director of the Institute for Financial Services Analytics

Approved: _____

Bruce Weber, Ph.D.
Dean of Lerner College of Business & Economics

Approved: _____

Douglas J. Doren, Ph.D.
Interim Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Bintong Chen, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Starnes Walker, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Michael Silas, Ph.D.
Member of dissertation committee

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Bad actors have embraced automation and current malware analysis systems cannot keep up with the ever-increasing load of malware being created daily. As a result, traditional malware detection and classification techniques using expert systems and brittle heuristics are outdated and ineffective. We introduce deep learning models based on inexpensive static features gathered from large scale malware datasets to generate robust and efficient malware detection and malware family classification predictions.

Static analysis is performed by dissecting or disassembling the malware's binary file and studying the components without executing it. Furthermore, static analysis is generally much faster than most malware analysis techniques. However, some static analysis of malware can be computationally expensive and not all static analysis should be considered for every sample in a large malware dataset. We introduce a meta-model trained using deep learning that finds the simplest classifiers to characterize and assign malware into their corresponding families. Using static analysis of malware, we generate descriptive features to be used in conjunction with deep learning, in order to predict malware families. Our meta-model can determine when simple and less expensive malware characterization will suffice to accurately classify malicious executables, or when more computationally expensive descriptions are required.

One of the most important components of training deep learning models, particularly deep neural networks, is finding the optimal model configuration and feature set combinations. Most applications of deep learning, specifically neural networks, use heuristics or trial-and-error to find the optimal model configurations. We implemented a large scale model configuration search using supercomputing resources to produce the most accurate deep learning model given a feature set. In addition, we construct a genetic algorithm used to find the optimal subset of static analysis features. This result provides us with the ability to construct extremely accurate deep learning models for malware detection and malware family classification.

Chapter 1

INTRODUCTION

Malware (i.e. malicious software) is software that is intended to damage, disable, or steal vital information from computer systems. Common examples of malware include computer viruses, worms, Trojan horses and ransomware [73]. For example, the WannaCry ransomware attack in May 2017 was an example of a randsomware attack which targeted computers running the Microsoft Windows operating system by encrypting data and demanding ransom payments [57]. Traditionally, companies protected themselves from malware using anti-virus programs such as Norton or McAfee. These programs work by creating a unique identifier (i.e. a hash) for each malware sample. This unique identifier is stored in a database that is pushed in patches to the antivirus program running on an end point. An end point is a device (i.e. laptops, tablets, mobile phones) connected to our organization's central network. These connections create attack paths for security threats. All incoming files on a user's computer can be checked against the database of the antivirus program that contains signatures of all known malware samples. One problem with this technique is that zero-day or unseen malware will not be in the antivirus' database and therefore will not be detected [21, pp. 34-44]. This problem is compounded by the fact that malware

creators have embraced automation to develop programs that can take a malware sample and automatically change it slightly with the effect of changing its unique identifier [34]. Automation tools can generate large numbers of malware variants and traditional techniques to detect malware no longer scale with the ncreasing the size of the threat landscape.

Current malware analysis systems tend to exhibit high detection rates for previously analyzed malware, for which signatures have been generated, but fail at detecting zero-day exploits for which malware is unavailable [12, pp. 110-115]. Therefore, the need for advanced techniques to be able to detect and classify malware is necessary. As cyber-attacks are growing in complexity and sophistication, the use of machine learning techniques has become indispensable for firms to become more efficient in recognizing patterns that constitute a risk to their information.

Creating a robust malware detection system requires extensive analysis of a malware sample. The two main ways of analyzing malware are static and dynamic analysis. Static analysis is a method of reverse engineering that is done by extracting a malware's code without executing the actual program [20, pp. 76-79]. An example of a static feature captured through static analysis could be computing the file's entropy or measuring the distribution of bits in the file. For example, one can predict with a high probablitiy that a file with high entropy is encrypted. Dynamic analysis refers to executing a file in an isolated environment or sandbox and monitoring its behavior to generate a report for further examination [86]. An example of a feature collected through dynamic analysis would be the list of web sites the file tries to access when

run.

The results of static analysis on the malware files will be characterized in terms of basic file information, byte level information, and the reverse engineered assembly code. These characterizations will make up the feature vectors we use to train our machine learning models. One of the characterization techniques we use is to represent the malware's executable code as a graph. We scan this graph in increasing granularity in order to characterize the important aspects of the malware's code. We use simple features such as the strings in the file, which can be an effective characterization technique [37, pp. 9-17]. These feature vectors can then be preprocessed in various ways to maximize the accuracy of the prediction models.

Malware detection is a binary classification problem where a model takes as input static features of a file and the model predicts if a file is malware or not [73]. Malware family classification is a multiclass problem where a model is used to predict what family a malware belongs to [5, pp. 3-14]. A malware family generally characterizes how a malware can infect computers what important capabilities a malware has and how to remediate that threat. This information is vital to a cybersecurity analyst.

In our research, we build highly accurate malware detection and family classification models using large scale datasets. Using the cloud platform we have built, we have analyzed the cost of producing various malware characterizations and have created models that keep time cost low while having high accuracy predictions. Our machine learning platform gives us the ability to create comprehensive models of our data and train them continuously or in batches until the desired accuracy is reached.

We explore supercomputing methods of searching for the best model configuration using all features to produce the highest accuracy prediction. Also, we use genetic algorithms to search for the best subset of features that can produce very high accuracy predictions.

The experiments that we discuss in this dissertation will make contributions to the current research landscape of malware detection and family classification using machine learning. We seek to overcome some of the challenges that exist in large-scale malware characterization and machine learning including how to scale static analysis to an extreme scale, creating cost efficient and effective deep learning models for detecting and classifying malware, and knowing which features and models work best for training on very large datasets.

Chapter 2

BACKGROUND AND RELATED WORK

In this chapter, we will describe background information and related work. We first discuss malware analysis which we use to create features. We then build datasets comrpised of these features to construct machine learning models. We describe our features in more detail and explain our cloud computing platform which will allow us to scale our analysis and machine learning process.

2.1 Malware Analysis

The first step in creating machine learning models is the creation of a dataset. The dataset is comprised of features extracted from the analysis of malware. Malware can be analyzed using two main methods: static and dynamic analysis. In the next sections we will describe static and dynamic analysis and how it is accomplished in our machine learning pipeline.

2.1.1 Static Analysis

Static analysis is a method of extracting a malware's code without executing the actual program and was first introduced in the field of malware detection in 1995 [20, pp. 76-79] [52, pp. 541-566]. In contrast to dynamic analysis which can take minutes to hours, static analysis of a malware can typically be performed in less than a second.

Automated tools like portable reverse engineering frameworks can help extract static features from a malware binary in a very accurate and effective manner, providing valuable insights about the intent of the suspicious file. Nevertheless, because of obfuscation techniques, static analysis alone is not enough to detect or classify malicious code [59, pp. 421-430]. We use Radare2 (r2), an open-source reverse engineering compiler, to extract static features to supplement our dynamic analysis [66]. Radare2 is designed as a lightweight tool to help disassemble software. It implements an advanced command line interface for moving around a file, analyzing data, disassembling, binary patching, data comparison, searching, replacing, visualizing and it can be scripted with a variety of languages, including Ruby, Python, Lua, and Perl. The reason we use this tool is that it is open-sourced, there is significant community support, and that static features can be extracted fast and cheaply.

In our pipline, our goal is to be able to analyze very large datasets upwards of one million malware samples. This scale of malware analysis lends itself to static analysis as it can be completed much faster and cheaper than dynamic analysis. However, we do potentially lose important information that could characterize the malware by not including dynamic analysis. We will discuss this further in Section 3.1.

2.1.1.1 Malware's Executable Code

Performing static analysis of an executable requires reverse engineering of the malware's executable code[1]. Figure 2.1 depicts how a call graph with features at each

[1]a set of instructions executed directly by a computer's central processing unit

node is constructed from the output of a disassembler. A call graph represents calling relationships between subroutines in a computer program. Each node in the graph represents a procedure and each edge indicates that the preceeding node calls the subsquent node.

First, the executable is disassembled into functions, blocks, and operations. Second, the disassembled code is transformed into a graph-like data-structure (e.g., call graph). Third, a feature graph is extracted from the graph. These features can then be used as input to our machine learning models.

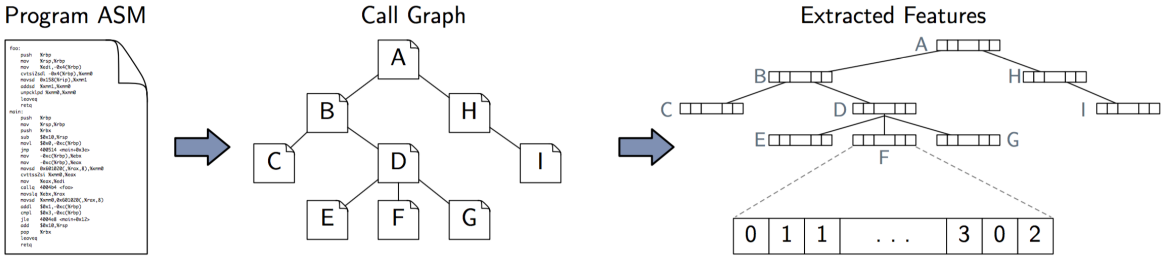

Figure 2.1: Extracting graphs and features from malware code.

2.2 Malware Datasets

Training a machine learning model is extremely dependent on the chosen dataset. Neural networks require huge datasets to train accurate models especially given a deep network with a large number of hidden layers. Due to the sheer number of malware samples we intend to analyze and train on, static analysis will be our primary method of analyzing malware. Contrary to static analysis, dynamic analysis requires a large amount of time to run. In this section, we describe the datasets we intend to build to solve the problems of malware detection and malware family classification.

2.2.1 Datasets

We will obtain up to a million malware samples from Reversing Labs [69] to be used as our training dataset. We obtained results on the malware family classification problem on a much smaller dataset of a few thousand samples. The largest curated malware dataset we have used consisted of 270,000 malware samples from 27 distinct families. However, we keep in mind that more malware is not necessarily better and that a curated, balanced dataset is what is most desired. For the malware detection problem, goodware will be gathered from a diverse set of Windows OS platforms including XP, 7, 8, and 10 for both 32-bit and 64-bit architectures. The goodware will include official Windows software pre-installed with the OS and popular open-source Windows applications.

2.3 Feature Characterization

In this section, we describe the feature characterization techniques used to represent a malware sample. The three main groups of features we use to characterize a malware are basic, byte, and assembly features. We will describe each group and the characterization techniques in detail. A table summarizing the groups and corresponding features can be seen in Table 2.1. A full feature description of all 47 final feature variations can be seen in Figure 5.5.

2.3.1 Basic

In this section, we present three static features we gather from the malware files also introduced by Saxe, et al. [77, pp. 11-20]. These features are constructed from

Static Feature Set Groups		
Group	**Features**	**Computational Cost**
Basic	Strings Metadata Import	1-5 seconds
Byte	Byte-Entropy Histogram	1 second
Assembly	Function Block Operations	Up to 30 seconds (with timeout)

Table 2.1: This table shows the static feature set characterizations and their shapes.

the list of strings included in the file, the metadata table, and the import table of the Portable Executable (PE) Header. The PE format is a data structure that encapsulates the information necessary for the Windows OS to load the executable code. The ASCII Strings are obtained using the `string` GNU tool. The python module `pefile` is used to extract the metadata and import tables from the PE Header. We create a fixed sized feature vector given either a list of ASCII strings or extracted import and metadata information from the PE Header. The arbitrary length of ASCII strings is transformed into a feature vector of length 256. First, each string is "hashed" to an integer between zero and the desired vector length. Second, an histogram of these hashes is produced by counting the occurrences of each values. The result is a vector of positive integers of the desired size. This is a best practice from Saxe, et al. [77, pp. 11-20], to be able to convert the variable string output into a fixed size array. Figure 2.2 shows how a list of strings is transformed into a fixed size feature vector.

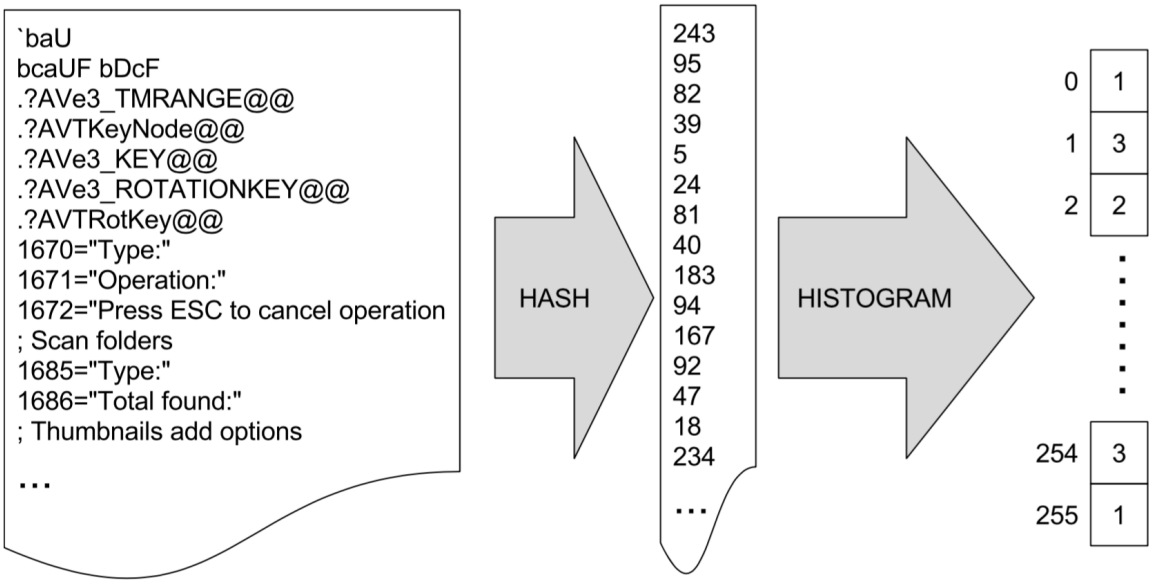

Figure 2.2: Feature extraction from malware file's strings

2.3.2 Byte

Byte-entropy histograms are an expressive representation of files and can be used as a state-of-the-art characterization of files suitable for deep neural networks [77, pp. 11-20]. To construct this characterization, we scan files using a sliding window of length 1024 with a step size of 256 bytes. We compute an entropy for each window using the histogram of bytes. Finally, histograms are accumulated in the 2D byte-entropy histogram in one of 256 entropy bins. The process of generating the byte-entropy histrograms is shown in Figure 2.3.

2.3.3 Assembly

One powerful method of characterizing malware is a assembly characterization. This technique corresponds to an expressive method of representing assembly code

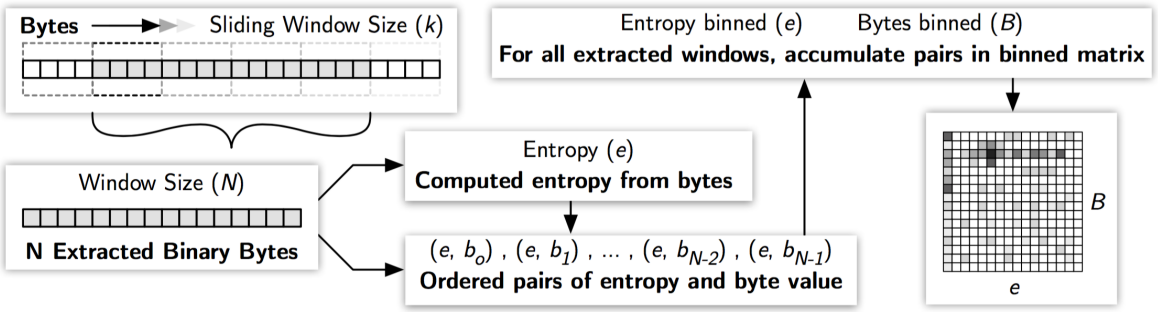

Figure 2.3: First, the file is scanned by a sliding window of length 1024 with a step of 256 bytes. Then, for each window, the bytes histogram is extracted and the associated entropy is computed. Pairs of byte and entropy are collected for all windows. Finally, the pairs are counted in the bytes-entropy histogram.

extracted from executables. We use Radare2 [66], a free and open-source disassembler, to analyze executable files. The advantage of Radare2 is that it "disassembles" many kinds of executables, including x86, ARM, Bytecode (Java), Javascript (from HTML files), etc. Figure 2.1 depicts how the call graph with features at each node is constructed from the output of a disassembler such as Radare2. We parse Radare2 output and extract the results of call and control-flow analyses into one data-structure depicted in Figure 2.4. This data-structure is a graph of operations linked by calls (curved arrows), branches (angled arrows), and fallthroughs (thin straight arrows).

To apply machine learning, we need to extract feature vectors from each node in the data-structure where each node corresponds to a block in the original code. We extract this representation by scanning this data-structure from operations to blocks to functions level. Along the way, we extract statistics about each level that we accumulate. This gives us three different levels of granularity.

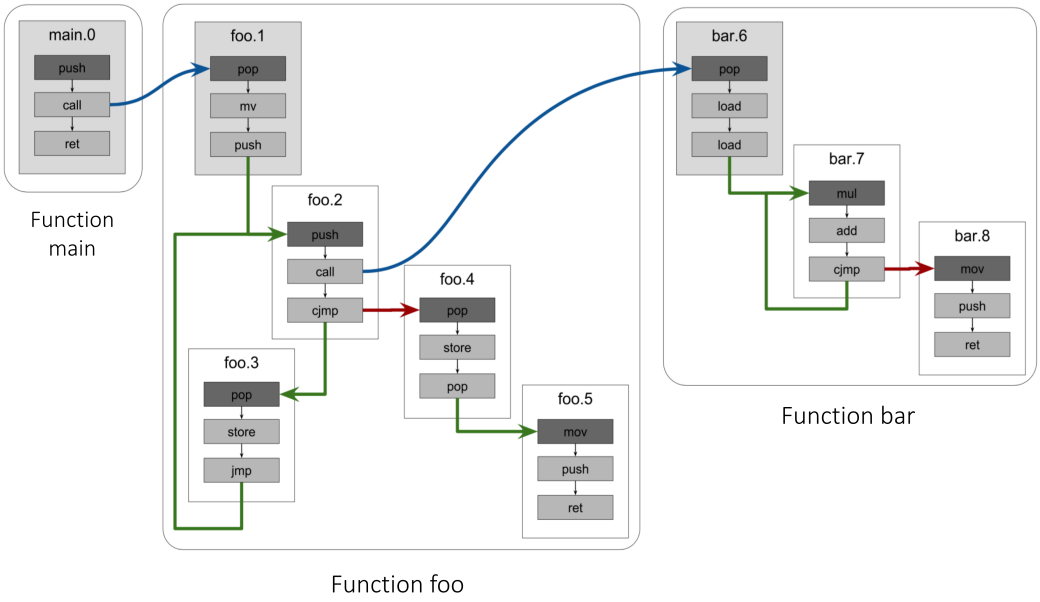

Figure 2.4: Depiction of code graph extracted from Radare2 output.

2.3.3.1 Operation Level

The whole-program instruction-flow-graph (WPIFG) connects operations using calls, branches, and fallthroughs. For each operation, one feature vector is generated containing statistics: operation kind and size.

2.3.3.2 Block Level

The whole-program control-flow-graph (WPCFG) connects blocks based on the calls and branches. Each block is characterized by two feature vectors: statistics and instruction 1-grams. The statistics include the block's size and edges statistics, but also aggregate the average statistics of its operations. The instruction 1-grams are histograms of the sequences of one operation presented as vector.

2.3.3.3 Function Level

The call-graph (CG) connects functions based on the calls. Each function is characterized by two feature vectors: statistics and instruction 1-grams. One function's statistics include the function's information and the average statistics of its blocks and operations.

2.3.3.4 Spectral Features

The three graphs (WPIFG, WPCFG, and CG) we collect during our disassembly analysis contain information characterizing malware at different levels of code. We use these graphs to create spectral features at the function, block and operation level. Each graph is saved in the form of an adjacency matrix comprised of nodes and edges. Depending on the graph, nodes are functions, blocks, or operations and edges are the connections between the nodes. Eigenvalues are generated from the laplacian of the adjacency matrix. The top twenty eigenvalues are saved for use in our deep learning models as they are an expressive characterization of the assembly graphs at different layers of code.

2.3.4 Feature Summary

We categorize our feature sets in three main groups: basic, byte, and assembly features. In Table 2.1 and Table 2.2, we summarize the feature sets considered in our experiments. In addition to the feature sets listed, we collect metrics during our analyses which we also use as input. These features are small in size, but contain valuable information. With so many features, we need to consider the possible preprocessing

transformations that could aid in the construction of our machine learning models. We will discuss this in the next section.

The computational cost of producing our features can be seen in Table 2.1. Basic features are generated by multiple tools running asynchronously where each analysis can take around one second. Byte features are generated using optimized C code and depend on the size of the file, but remains extremely efficient. Assembly features take the longest to produce as we use Radare2 to conduct a number of analyses. We currently have a timeout of 30 seconds for assembly features to limit the runtime of certain analyses that can take hours or even days to complete.

Static Feature Sets			
Characterization		Format	Size
Basic	Strings	vector	256
	Metadata	vector	256
	Import	vector	256
Byte	Histogram	vector	256
	Byte-Entropy Histogram	matrix	[16, 256]
Function	statistics	matrix	[20, 19]
	1-grams	matrix	[20, 54]
	eigenvalues	vector	20
Block	statistics	matrix	[20, 8]
	1-grams	matrix	[20, 54]
	eigenvalues	vector	20
Operation	statistics	vector	20
	1-grams	matrix	[20, 54]
	eigenvalues	vector	20

Table 2.2: This table shows the static feature set characterizations and their shapes.

2.3.5 Preprocessing

After the raw features are generated using malware analysis, we consider transformations to aid in training of our models. Certain feature sets are more effective after being transformed rather than using the original representation. Normalization has proven very effective at removing the machine learning model's ability to learn the invariants of the dataset instead of the key features. Absolute values of features could be important, but small values may be lost when the whole dataset is normalized. To prevent this, we also present to the neural network logarithms of each feature. Our three preprocessing techniques we use are as follows:

- ID - identity, no preprocessing
- log - compute the logarithm of each element
- norm - normalize the tensor[2] (Can provide row, column, or global normalization)

2.4 Deep Neural Networks

Deep Learning is a subfield of machine learning concerned with algorithms inspired by the structure and function of the brain called artificial neural networks. In this section, we will present neural networks, how they are trained, our method of implementation, and how we compare models.

[2] a multidimensional array

2.4.1 Artificial Neural Networks

Artificial Neural Networks (ANNs) are models made up of interconnected artificial neurons. A model of an individual neuron can be seen in Figure 2.5. The output of this single neuron model is therefore:

$$y = \sigma(w_0 * x_0 + w_1 * x_1 + w_2 * x_2 + b),$$

where x_i are the inputs of the neuron, w_i are the respective weights for each of the x_i, σ is the activation function, and b which is called the bias. The activation function is typically a sigmoidal function such as the hyperbolic tangent (tanh) or the logistic function.

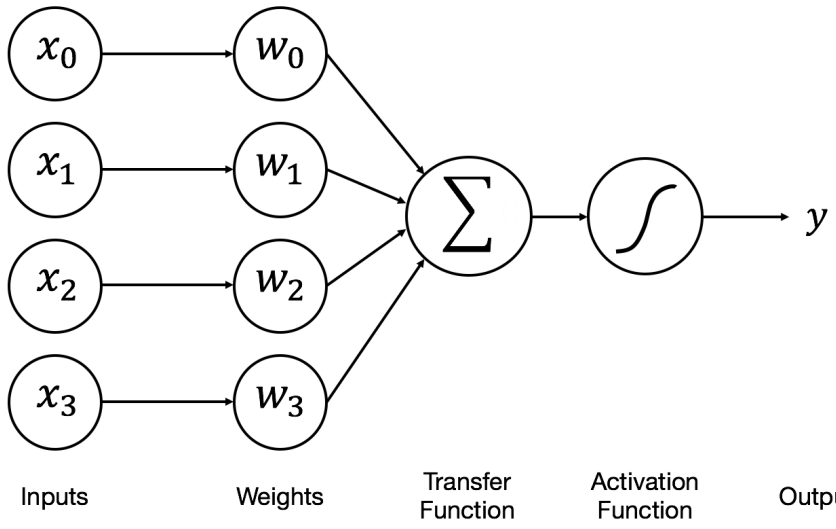

Figure 2.5: This figure depicts an individual neuron in a neural network structure.

Artificial neural networks (ANNs) can be viewed as weighted directed graphs in which the nodes comprise of neurons and the edges are conections between the neurons.

Based on the graph's architecture there exist two main categories:

- *feed-forward* networks
- *recurrent* networks

In feed-forward networks, or in our case multilayered perceptrons, the neurons are connected in one direction and contain no loops. In feed-forward networks information flows in one direction and samples are independent of each other which in a sense makes this network memory-less. On the other hand, recurrent networks have feedback paths in which the input to the neuron is modified with each sample. In this paper, we focus on feed-forward networks or multilayered perceptrons (MLPs).

MLPs are typically *fully-connected* where all outputs of one layer are inputs to the next layer. An example of an MLPs structure can be seen in Figure 2.6. The strucuture of an MLP starts with an input layer of fixed size, a hidden layer which can be comprised of multiple layers, and an output layer which holds the result of the network. MLPs are incredibly powerful and Hornik et al. [31, pp. 251-257] [32, pp. 359-366], has proven that given a sufficient number of layers with non-linear activation functions, an MLP can approximate any (measurable) function.

2.4.2 Deep Learning

In this section, we discuss how neural networks are trained. Neural networks are trained with a goal of producting a network that correctly predicts the target function

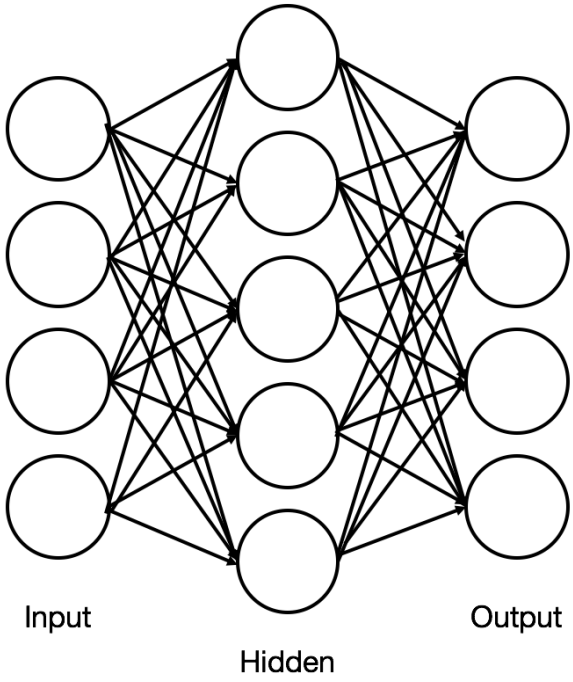

Figure 2.6: This is the general structure of a multilayered perceptron.

$\phi : \mathbb{R}^n \to \mathbb{R}^m$. The training set $\alpha \subset \mathbb{R}^n$ is comprised of input samples $x \in \alpha$ where $\phi(x)$ is known for all samples. The function realized by the network we denote as $f(x)$.

The most popular neural network algorithm is the back-propagation algorithm proposed in the 1980's [72, p. 1] and is normally used in conjunction with an optimization method such as gradient descent. The goal is to find the values of the parameters Θ that minimize a loss function $\mathcal{L}$. It is common to use the L2 norm as the loss function, e.g., $\mathcal{L}(x) = \|\phi(x) - f(x)\|^2$. The back-propagation can be seen below:

1. **Forward propagation**: Compute the loss function for all nodes in the network.

$$\mathcal{L}(x) = \|\phi(x) - f(x)\|^2$$

2. **Backward propagation**: Compute $\frac{\partial \mathcal{L}}{\partial \rho}(x)$ for all parameters ρ in the network.

3. **Update**: For all $\rho \in \Theta$,

$$\rho \leftarrow \rho - \ell * \frac{\partial \mathcal{L}}{\partial \rho}$$

where $\ell \in \mathbb{R}^{*+}$ is the *learning rate*.

The back-propagation algorithm starts with evaluting the output at each node for a given sample. The backward propagation step evaluates the influence of each parameter ρ on the loss function. The update step is based on gradient descent in which the parameters ρ are updated. Each sample in the training set will use the back-propagation algorithm to update the parameters of the network. One pass through the training set is denoted as an *epoch* and many epochs are required to train a network.

The learning rate, ℓ, the number of layers, size of the layers, and the activation function of each layer are part of the hyper-parameters of the network. Hyper-parameters affect the convergence of the learning algorithm. These parameters cannot be directly estimated from the data and therefore must be chosen by the practitioner. Methods to automate this tuning process without domain knowledge include grid search [10, pp. 153-160], random search [11, pp. 281-305], bayesian optimization [25, pp. 3460-3468], covariance matrix adaptation evolution strategy using GPUs [53], and non-probabilistic radial basis function surrogate model [35].

Deep neural networks are neural networks comprised of many layers of neurons. The idea of deep neural networks was introduced soon after neural networks [38] [39, pp. 364-378]. Around the turn of the century, the interest in deep neural networks was

newed with the increase of compute capabilities with tools such as cloud computing and GPUs.

2.4.3 Theano: Our Deep Learning Implementation

Our experiments are conducted using Theano which is a numerical computation library. We have build our own deep learning framework around Theano to implement deep learning algorithms and help facilitate training our deep neural networks. Our framework does not consider any specific use-case and provides low-level primitives. By doing so, it permits us to define any neural network we might need. In addition, this framework has been constructed to scale infinitely for very large datasets on distributed memory systems.

2.4.4 Machine Learning in the Cloud

Deep neural networks have proven to be a powerful method of solving hard problems, but when coupled with large amounts of training data, it has also proven to be computationally intensive. Powerful hardware is necessary for training complex neural networks with large amounts of data, and using cloud-based services is a cost-efficient way of achieving this. In the case of model and feature set exploration, many models have to be trained simultaneously and the best way to do this is to scale the number of cloud instances to fit our compute requirements. In Section 5.1, we leverage supercomputing resources to help facilitate our computation needs.

We have built a scalable cloud platform to complete the malware analysis discussed in Section 2.1. For static analysis, cloud instances are initialized which continuously poll for analysis requests. Once a request is made, our software proceeds to download malware binaries, conduct static analysis, and submit the results to a cloud data store. For dynamic analysis, we have created a scalable platform in which malware can be submitted for analysis in a sandbox environment hosted on the cloud. The results of the analysis are submitted to the same cloud data store.

Our platform is able to track the cost of analysis, the time required to analyze malware samples, and much more. Also, we can track the time and cost of producing our deep neural network models. We can use this information to compare the cost of producing the dataset, the cost of training the model, and the resulting accuracy. In a realistic environment, models need to be created per the business requirements and should take into account the dollar and time cost of producing a model.

2.4.5 Comparing Models

When machine learning is used to solve a task, multiple algorithms are used to build various models until the best model is found. To construct the best model for a specific task, we generally evaluate the ability of these models to generalize what was learned on a training set to predict targets given a testing set. This process requires both a proper experimental setup and a measure of the accuracy of a given model. We use (n-fold) cross-validation and different measures of accuracy.

2.4.5.1 Cross-Validation

Cross-validation is a model validation technique which tests how well the model performs and is able to generalize to an independent dataset. The process of cross-valiating is to partition the dataset into complementary subsets performing the model training on one subset and validating the model with the other. Multiple rounds of this are performed with different subsets and the validation results are averaged to find a final prediction estimate.

Typically, we use n-fold cross-validation in which the dataset is split into n subsets of the same size. Cross-validation is performed n times and the validation accuracies are averaged for a final result. Zeng et al. [90, pp. 1-12] proposed a method that provides balanced intraclass distributions when partitioning a dataset into multiple folds. This method uses stratification, which means that all folds contain the same number of instances for each classification target, giving them the same class distribution as the original dataset. This method of cross-validation is what we use for all our experiments.

2.4.6 AWS Cloud Infrastructure

Our machine learning platform is hosted on Amazon Web Services (AWS) where we do most of our training, dataset creation, and analysis of malware. AWS offers reliable, scalable, and inexpensive cloud computing services as well as well as many storage services. The ability to add compute power on-demand for training models or analyzing malware is powerful. The four main services we work with are:

- Elastic Compute Cloud (EC2) with Auto Scaling Group (ASG) to host the applications in a scalable way,
- DynamoDB to store the descriptions of the models, datasets, and to record accuracy metrics while training models,
- Simple Storage Service (S3) to store the datasets and the parameters of the trained models, and
- Simple Queue Service (SQS) to distribute jobs.

In Figure 2.7 depicts the process of training our machine learning models. Spot instances are used to reduce the cost of training models. Spot instances allows users to bid on spare Amazon EC2 computing capacity. Since Spot instances are often available at a discount compared to On-Demand pricing, we can significantly reduce the cost of training models. These instances are self-contained compute machines that pull from the job queue, load the dataset and model and produce results. These results are then stored in our S3 buckets and DynamoDB. A bucket is a unit of storage in AWS object storage service S3. Amazon's DynamoDB acts as a fast and flexible NoSQL database service for storing our results.

2.5 Literature Overview

Malware analysis is a necessary step in generating data and therefore features to train machine learning classifiers. A subset of the best analysis methods, tools, and techniques were surveyed by Uppal et al. [83, pp. 103-120]. Static analysis techniques

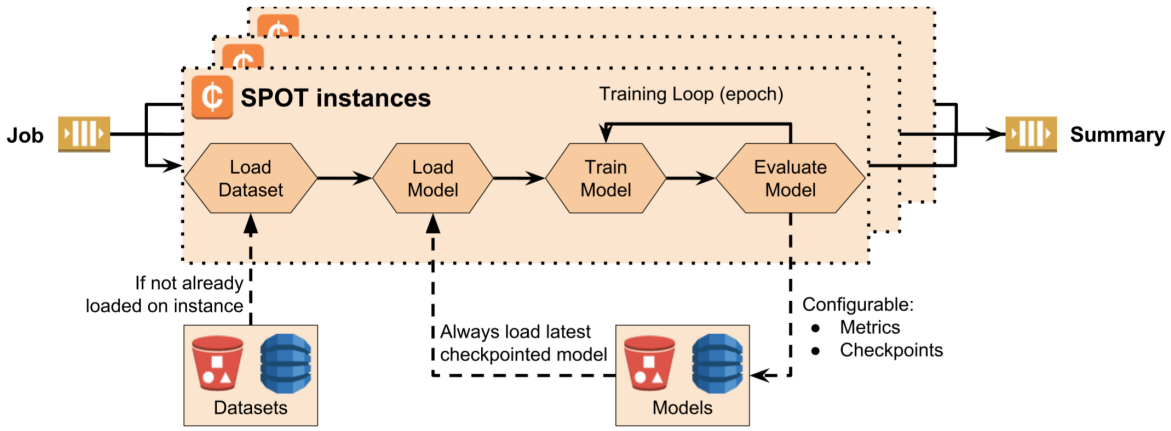

Figure 2.7: This figure shows our configurable AWS platform we have built to train our models.

have also been recently surveyed and analyzed [61, pp. 440-450]. A dataset of Android malware was used in conjunction with static analysis to train and produce accurate malware detection models [74, pp. 141-147] [6]. Shabtai et al. [79, pp. 16-29] trained many machine learning classifiers using static analysis as input to conduct malware detection. However, static analysis has limitations such as when malware exhibits code obfuscation techniques or if it is packed [59, pp. 421-430].

The use of Cuckoo [22] to conduct dynamic analysis has been confirmed as an effective malware analysis tool [65, pp. 225-236] [85, pp. 1-6]. In addition, the efficiency of dynamic analysis has been improved by leveraging the fact that most malware samples are variants of existing malware [8, pp. 1871-1878]. Dynamic analysis can be used in conjunction with Support Vector Machines for the detection of malware [4, pp. 247-258] [29, pp. 45-54]. Firdausi et al. [26, pp. 201-203] conducted a survey of machine learning techniques used to build malware detection models. Dynamic or behavioral analysis has also been used to cluster malware families [71, pp. 639-668].

Most malware detection classification models only use static or dynamic analysis, but not both. However, Spreitzenbarth et al. [81, pp. 141-153] created a sandbox that combines both static and dynamic analysis. Also, Santos et al. [27, p. 56] implemented a method to use both static and dynamic analysis features to detect malware at a high accuracy. The use of machine learning for the detection and classification of malware is expansive and has been surveyed at length [47].

Malware detection research is even more necessary as millions of malware are being introduced into the wild every day. Many machine learning classifiers are trained using static analysis to detect malware including Bayesian classifiers, rule-based methods, decision trees and ensemble methods to produce highly accurate models [3, pp. 55-62] [75, pp. 64-82] [89, pp. 313-320] [87, pp. 25-36]. In addition to static analysis, detection models can be made using dynamic analysis features using classifiers including Random Forest, K-Nearest Neighbor and many other machine learning algorithms [1, pp. 86-103] [80] [88, pp. 37-42] [14, pp. 13-20] [42]. Anderson et al. [5, pp. 3-14] bridged the gap between static and dynamic analysis by combining them into a singular model. However, all of these studies have been on relatively small datasets that only reach tens of thousands of malware samples. Yerima et al. [77, pp. 11-20] conducted a study using static analysis and machine learning to create an accurate model using over 400,000 malware samples. Dahl et al. [23, pp. 3422-3426] used a proprietary Microsoft platform to analyze 2.6 million malware to create a 99.5% accurate malware detection system.

In addition to experiments that produce high accuracy malware detection models, many researchers have looked at specific difficulties with the robustness of these

models. Demontis et al. [24] analyzed Android malware detection models ability to detect evasive malware and created a platform to mitigate malware evasiveness. In addition, Chen et al. [19] investigated the robustness a cyber system to learn to detect new malware over time. Techniques for determining malware variants have been examined by Cesare et al. [16, pp. 181-189] and for determining malware similarity using parallelization [78, pp. 69-77]. Specifically for malware detection, feature selection has been scrutinized both experimentally and heuristically [9, pp. 113-120] [41].

Malware family classification is a difficult problem as the input to the models are similar to the detection problem, but the predicted output has to include many different malware families. Similarly to malware detection research, both static and dynamic analysis features have been explored as input to machine learning models. Research involving static features include using Opcode sequences and clustering for classification [62, pp. 95-107], using function and printable string information [37, pp. 9-17], and using N-grams for classification [76, pp. 251-256]. In addition to static analysis, Ahmadi et al. [2, pp. 183-194] includes expert analysis to machine learning model increasing accuracy to over 99%. Dynamic features is used in conjunction with clustering to generate separate behavioral families of malware with impressive results [56, pp. 251-266]. The use of both static and dynamic features in a classification model is applied on small datasets of a couple thousand samples [36, pp. 646-656] up to the hundreds of thousands of samples [50, pp. 422-433]. For malware that is packed or polymorphic, malware classification and detection is more difficult as analysis techniques may not generate helpful features. Cesare et al. [17, pp. 1193-1206] creates a classification

model with this in mind. Converting static analysis results into images has proven to be an effective input for malware classification methods as well [60] [28]. The speed of malware classification was reviewed by Moonsamy et al. [58, pp. 176-188] who proposed feature reduction to classify malware faster.

Feature selection and model exploration has to be considered in order to build the best neural networks for a given problem. In the context of cybersecurity, the number of features that can be generated by an individual malware sample is enormous. The best feature selection methods have been surveyed for a general problem [18, pp. 16-28] and on selecting the most relevant features and examples from the dataset [13, pp. 245-271]. Ranveer et al. [68] tests the best feature selection methods in the context of malware detection, but only for a subset of the potential static features.

Our research extends the current literature landscape by creating a feature set that is well beyong the breadth of any related research in both malware detection and malware family classification. We introduce a novel cost-based deep learning model that focusses on speed and accuracy where most research focusses solely on accuracy. Furthermore, we introduce an exhaustive model search using supercomputing resources and a genetic algorithm for feature selection to produce accurate models using a much more robust feature set than the current research exhibits.

Chapter 3

MALWARE FAMILY CLASSIFICATION

In this chapter, we will discuss our results for the malware family classification problem. Malware can be categorized into a malware family which is a grouping of malware based on common characteristics. If the malware family of an unknown malware sample can be predicted, so can the characteristics and important capabilities of that malware. For example, if an unknown malware sample is predicted to be a member of the malware family Zbot, we can predict that the malware's main focus is on stealing sensitive online banking information, e.g., credit card numbers, pin codes, and passwords. This information can be used by security analysis for further remembiation of the malware since vital information of the malware would be known such as its threat level, contagion danger, and capabilities. In the next section, we describe an experiment of classifying a dataset of malware into their corresponding malware families.

3.1 Dynamic Analysis

Dynamic malware analysis has emerged as the state-of-the-art detection approach that compensates for the shortcomings of signature-based techniques [86]. In this type of analysis, a malicious file is executed in an isolated environment or sandbox,

and its behavior is monitored and reported for further examination. While this technique is a much more robust method of identifying malware than traditional signature-based techniques, dynamic analysis still suffers from limitations due to the ability of anti-sandbox malware to detect the presence of the monitored environment, and to stop execution or perform benign activities if a sandbox is detected [45, pp. 287-301]. It also takes longer to perform dynamic analysis than signature-based techniques or static analysis. This barrier becomes even more dramatic, as malware authors continue to add logic to detect whether their malware is being executed in virtualized environments, hindering the ability of dynamic analysis systems to detect certain malware as a threat. We study the additional information dynamic analysis provides to the problem of malware family classification. These results show that dynamic analysis does improve performance and should be used. However, a tradeoff has to be considered between the cost and time resources of dynamic analysis because it is far more expensive and time consuming than static analysis. A depiction of the dynamic analysis pipeline can be seen in Figure 3.1. First, we launch EC2 spot instances that contain custom built Windows images that can communicate with our Cuckoo server. Cuckoo is an advanced, extremely modular, and 100% open source automated malware analysis system [22]. Cuckoo is used as our dynamic analysis engine as it is the number one malware analysis system used for research and it has significant community support. The output of Cuckoo analysis generates a behavioral report of the malware contained in a JSON object. Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems. The API calls of the malware

are then extracted and then converted into a histogram for use as a feature.

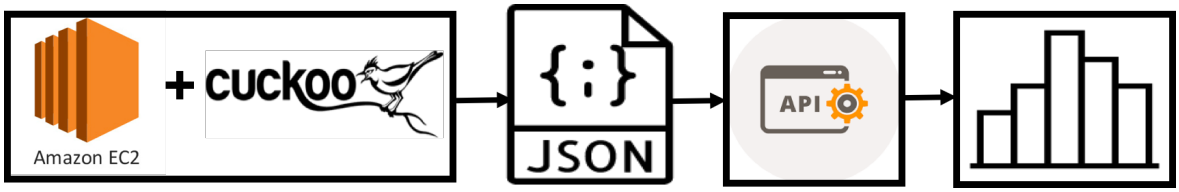

Figure 3.1: Depiction of the dynamic analysis workflow.

3.2 Malware Family Classification

An evalutation of the malware family classification model was conducted on a small dataset [43]. For this dataset, we prepared 3320 malware samples for characterization using static and dynamic analysis. We used Radare2 for static analysis and Cuckoo for dynamic analysis as described in Section 3.1. The figure below shows the static features extracted from the static analysis results. These four features (byte features, PE import features, string features, and PE Metadata features) are a subset of the features described in the feature characterization section (Section 2.3). The dynamic features that are used to build our machine learning models correspond to API calls from all malware seen in the dataset, which are generated from dynamic analysis. In addition, each of the malware's dynamic features is comprised of an API call histogram of 301 dimension. These features are concatenated into a hybrid feature set which we used to train a model as shown in Figure 3.3. We then evaluate the model with a dataset of unseen malware as shown in Figure 3.4.

Our hybrid malware family classification model is trained and validated using 10-fold cross validation and can be seen in Figure 3.4. A separate test set is used to

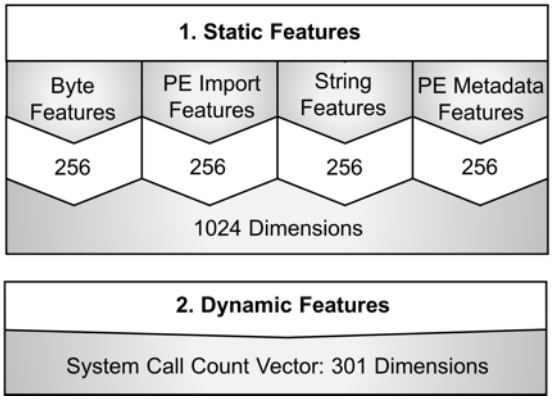

Figure 3.2: This figure shows the breakdown of static and dynamic features extracted from the analysis stage.

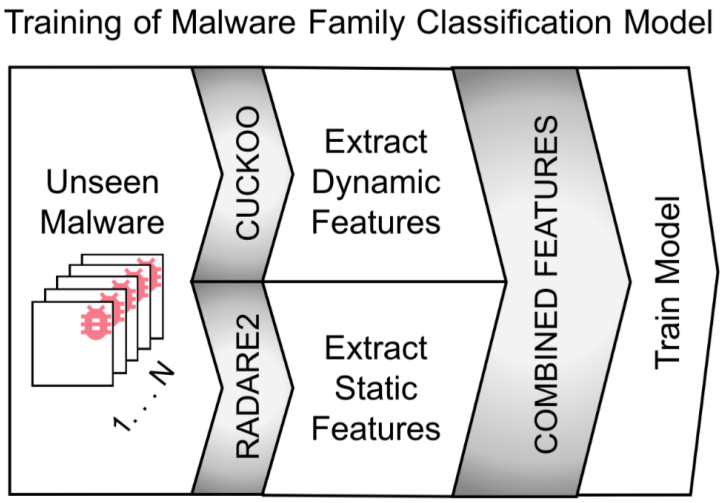

Figure 3.3: This figure shows the process of training the Malware Family Classification Model.

test the accuracy of the model. We test the accuracy of several models shown below with the highest accuracy being neural networks. The accuracy of the model is around 92% which is good for only using 3320 samples with a small number of features. Table 3.1 shows the results of the many machine learning algorithms we evaluated.

The confusion matrix for this experiment is shown in Figure 3.5. The matrix

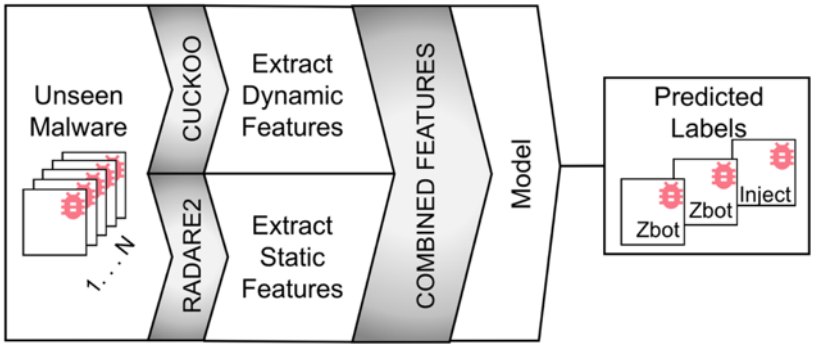

Figure 3.4: This figure is the malware family classification model, which is trained on both dynamic and static features combined into a hybrid feature set.

Table 3.1: Results for Malware Family Classification Model

Classifier	Accuracy	Precision	Recall
Neural Network	92.18%	89.19%	86.87%
Random Forest	91.70%	89.60%	84.12%
Support Vector Machines	91.58%	88.21%	85.49%
Decision Tree	89.31%	82.53%	81.66%
K-Nearest Neighbors	88.78%	81.67%	78.84%

shows that there were very few misclassifications and the size of the dataset seems to point to a lower accuracy than what was actually achieved. The following work for the malware family classification model will expand the dataset, the number of feature sets, and will focus on deep supervised learning using neural networks. Creating a large scale experiment with a focus on building deep neural networks provides us with a much more accurate model.

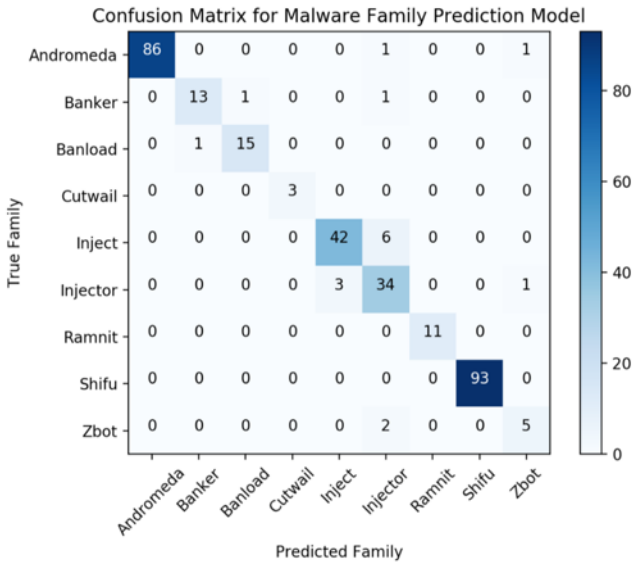

Figure 3.5: This figure shows the confusion matrix for the Malware Family Classification Model

Chapter 4

EFFICIENT CLASSIFICATION OF MALWARE USING DEEP LEARNING

In this chapter, we use deep learning to build a meta-model that finds the simplest classifiers to characterize and assign malware into their corresponding families. Using static analysis of malware, we generate descriptive features to be used in conjunction with deep learning in order to predict malware families. Our meta-model can determine when simple and less expensive malware characterization will suffice to accurately classify malicious executables, or when more computationally expensive descriptions are required. Finally, our meta-model is able to predict the simplest features and models to classify malware with an accuracy of up to 90%.

4.1 Motivation

Deep learning has recently emerged as the state-of-the-art technique for malware detection and classification, due to increasing computational capabilities and extensive datasets [77]. It has been proven that deep learning models can help analysts achieve breakthrough results in terms of both high accuracy, and low false positive rates for malware characterization [84] and classification [46].

To train these deep learning architectures, relevant features must be extracted from the code of malicious executables. One of the most widely used techniques for analyzing malware and extracting relevant features corresponds to static analysis. Static analysis is a method of reverse engineering that extracts code from a malicious file without executing the malware binary [44]. Moreover, static analysis is orders of magnitude faster than other techniques used for analyzing malware such as dynamic analysis [51]. In addition, static techniques observe the entire structure of malware and characterize all possible execution paths of a malicious sample, while dynamic malware analysis is limited to a single execution path that was executed by the program [61].

On the other hand, static analysis approaches are not without limitations of their own, i.e. it is well-known that they suffer from packing and obfuscation. However, several cybersecurity companies, such as Reversing Labs, have developed automated analysis technologies to remove all packing, obfuscation, and protection artifacts from malicious binaries and extract all internal objects with their metadata [70]. These unpacked objects can be used for further analysis and feature set extraction using different disassemblers.

Nevertheless, finding the set of static features that are relevant for the classification of malware is a complex task. In addition, extracting static features can be computationally expensive. Figure 4.1 shows the cost of producing three static characterizations (bytes, basic, and assembly) for datasets of varying sizes.

Byte-level analysis considers the raw bytes in malware binaries, and can be used to generate bytes-entropy histograms to identify the amount of information associated

with various bytes distributions in malicious files. Although byte features usually depend on the size of the file, they remain extremely efficient and take less than a second to produce, even for significantly large files. Basic features, such as the list of strings included in a malicious executable and the import table of the Portable Executable (PE) header, can normally be generated by multiple tools running concurrently, but take longer to produce than bytes features. Assembly features characterize a malware's disassembled code into graph-like data structures such as call and control flow graphs [84]. Feature vectors are extracted from different granularities of the code, i.e., operations, blocks and functions, resulting in three different levels of granularity that we can use to characterize a malware's code. These features, however, take the longest to extract and should not be used for every sample of a malware dataset. As a result, it is important to evaluate the cost of generating these features versus the accuracy of the deep learning models that can be constructed using different static characterizations of malware.

We use deep learning to construct highly accurate malware family classification models for a large malware dataset. Using the cloud platform we discussed in Section 2.4.6, we analyze the cost of producing various static characterizations of malware versus the accuracy of our models. The results of static analysis on the malware files are characterized in terms of basic file information, byte level information, and reverse engineered assembly code. These characterizations make up the feature vectors we use to train our deep learning models. Given that generating assembly features are computationally expensive, we construct a meta-model that identifies when simpler

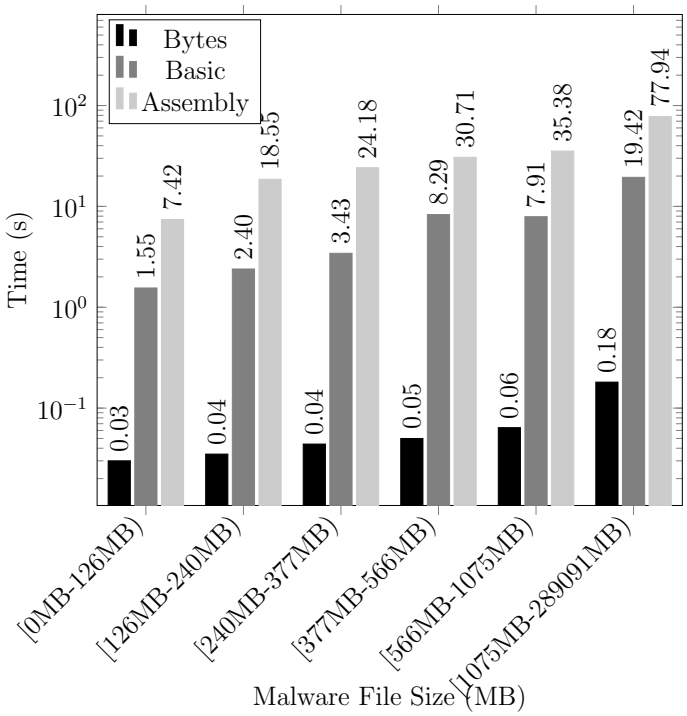

Figure 4.1: This figure depicts the cost of producing different static characterizations of malware. Analysis of malicious code at the byte-level is extremely efficient and takes less than a second to generate. Producing basic features, such as strings, normally requires more time than bytes features. Assembly features extracted from call and control flow graphs take the longest to produce, and are one of the most expensive static characterizations of malware.

and less expensive characterizations, such as byte features, are enough for correctly classifying malicious samples, and determine when basic and assembly features are required to produce a better prediction. By constructing a set of features of increasing cost and predictive power, we can reduce the overall time it takes to generate features for a large malware dataset, as we only generate the most costly features for a small subset of the entire malware corpus. Our results show that our "meta-model" is able to predict the malware families with an accuracy of up to 87%, using only the simplest and most inexpensive features for most of our samples.

4.2 Methodology

In this chapter, we develop models that are able to accurately classify malware into their corresponding families. These models are built using a variety of static analysis features that characterize files in terms of bytes, basic, and assembly features. The goal of our research is to construct a meta-model that accurately predicts which malware family prediction model to use to classify a specific malware sample, with the shortest execution time.

4.2.1 Training

We design our experiments to create a meta-model that selects the simplest model to use between three malware classifiers constructed using our three static feature sets. All our models can be used to correctly predict the family to classify a specific malware sample. We show that our meta-model obtains the best accuracy of any of our models, with an execution time that is comparable to the least expensive models.

4.2.1.1 Malware Classification Models

We construct three different malware family classification models trained using input from the three different malware characterizations. We name these three models after the corresponding static features they are trained on: Bytes (B), Bytes-Basic (BB) and Bytes-Basic-Assembly (BBA). We trained our models using deep neural networks (DNNs) on each of the static feature sets, since deep learning scales much better than other machine learning techniques when the size of our data grows to millions of malware.

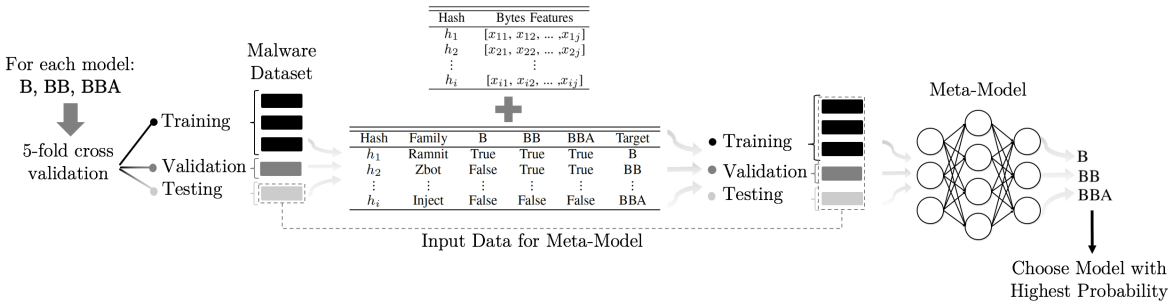

Figure 4.2: The left-hand side of this figure shows the general architecture of our malware classification models: Bytes (B), Bytes-Basic (BB) and Bytes-Basic-Assembly (BBA). For each of the three models, we split the dataset into five stratified folds: three of which are used for training, one for validation and one for testing. The cross-validation process is repeated five times and the results from the testing folds are used as input to the meta-model. Additionally, the right-hand side of this figure shows the pipeline for our meta-model. The meta-model's input features correspond to a feature vector of bytes, and the targets are one of the three classification models, Bytes (B), Bytes-Basic (BB) or Bytes-Basic-Assembly (BBA). The input data is again split in five folds: three for training, one for validation, and one for testing, and the cross-validation process is executed five times. The test set then predicts which malware family classification model to select (B, BB, BBA).

Figure 4.2 shows the deployment stage of our deep learning platform. We first split the dataset into five stratified folds. We then train the B, BB, and BBA models with three of the folds, while selecting a random fold as a validation set. The models are then used to predict the malware families of our samples in the remaining fold. The cross-validation process is then repeated five times, with each of the five folds used exactly once as a testing fold.

We choose to use 5-fold cross validation and break down our malware dataset into the aforementioned folds (three for training, one for validation, and one for testing) to ensure that the reported accuracy is representative of the models ability to generalize what is learned from the training set. It also guarantees that all observations are used for both training and validation, and each sample is used for testing exactly once. The

results from the five testing folds are then used to create the input dataset for our meta-model.

For each of the samples in our malware dataset, we gather information regarding whether our three models were able to produce an accurate classification. A correct prediction is denoted as "True", whereas an incorrect classification is labeled as "False". If a malicious sample is correctly classified by more than two models, we choose the simplest and most inexpensive model that yielded a correct prediction. For instance, if all B, BB and BBA models generate a correct classification, we choose B, i.e., Bytes, as the target for our meta-model training dataset because the static analysis to construct B is computationally less expensive than BB and BBA.

4.2.1.2 Meta-Model

The evaluation results of the malware family classification models are used as input to our meta-model. Given a malware sample, our meta-model should choose the most cost efficient model that still produces a correct prediction. If the evaluation is that no model produces a correct family classification, the Bytes model will be chosen.

The right-hand side of Figure 4.2 shows the architecture of our meta-model. The input vectors are byte features extracted from malicious binaries. The targets correspond to three classes: Bytes (B), Bytes-Basic (BB), Bytes-Basic-Assembly (BBA), indicating the classifier required to correctly categorize a malware sample. 5-fold cross validation is used to assess the effectiveness of our meta-model, where three folds are used for training, one for validation and one for testing. The test set predicts which

model to choose (B, BB, BBA) to classify our malicious executables into their corresponding families.

4.2.2 Model Configuration

To build our malware family classification models and meta-model, we evaluated different deep feed-forward neural networks with varying depths and structures using Theano. These models were trained for over two hundred epochs, and their results were compared in terms of error rate. Our preliminary results showed that several models took more than one hour to perform one epoch of training, rendering them impractical for our large-scale experiments. Even after several days of training, the error rates yielded by certain models was still much higher than the one obtained by much simpler architectures.

The configuration that produced the best results corresponded to a deep neural network model consisting of five hidden layers (512,512,128,128,64), with a Rectified Linear Units (ReLU) activation function for all hidden layers. It has been demonstrated that ReLU activation functions can speed up learning in the initial training stages, by maintaining a stable gradient descent convergence rate during the first sessions or iterations of the model [77]. For the output layer, we use the softmax function to ensure that the output of the model is a probability distribution. The softmax function (also called normalized exponential function) compresses a vector of arbitrary real values into a probability distribution.

4.2.3 Dataset

We obtained a malware dataset from Reversing Labs pertaining to a stream of malicious executables targeting financial institutions. These malware come from forty families, designed to infect a variety of MS Windows versions including XP, 7, 8, and 10 for both 32-bit and 64-bit architectures. Furthermore, the dataset contained samples that were gathered over a span of twelve years (2006-2018), with most of the files being collected in 2014 and 2016. For the experiments presented in this paper, we randomly subsampled each family with more than one thousand files, and used the maximum number of available samples per family. This left us with a dataset comprised of nine different families. Table 4.1 shows the breakdown of our dataset.

Furthermore, our dataset was curated to guarantee the extraction of relevant information from the malicious files. It is a well-known fact that most of the Windows malware are packed [82], and static analysis approaches fail at extracting meaningful features from malware. Our dataset provider removed all packing, obfuscation, and protection artifacts from the binary files to extract all internal objects with their meta-data. As a result, the unpacked malware were available for further analysis using our disassemblers.

4.3 Results

In this section, we present the major results from analyzing our malware dataset of over 100000 samples using our deep learning platform. As indicated in Figure 4.3, the malware classification model based on all of our static features (BBA) produced

Table 4.1: Composition of Malware Dataset

Name	Type	Count
Andromeda	virus	2222
Shifu	spyware	2257
Cutwail	downloader	3509
Banker	spyware	14596
Banload	downloader	15483
Inject	virus	15483
Injector	trojan	15483
Ramnit	trojan	15483
Zbot	downloader	15483

the highest accuracy (90.07%). However, it also required the longest amount of time to be trained and validated (41.67 hours), because that model uses all three static characterizations extracted from the malware. On the other hand, the model based on bytes (B) features, the least expensive characterization of our malware dataset, was able to produce modest results for the classification of malware. It achieved an overall accuracy rate of over 86.36%, and also took the least amount of time to train and validate out of the three classification models. This shows the high predictive power of bytes features, as this particular static characterization of malware holds enough information to accurately classify many malware binaries, which can be accomplished under a reasonable amount of time (17.28 hours).

In addition, we computed the accuracy, precision, and recall for each malware family in order to analyze the individual performance of our classifiers across class labels. As shown in Figure 4.4 and Table 4.2, the results generally improve as we use the most computationally expensive classifier (BBA) for most of our malware families. However, this model is not necessarily the best when predicting malware as being part

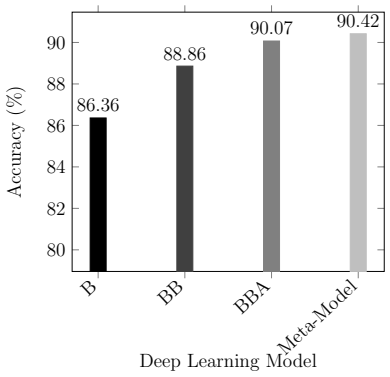

Figure 4.3: Accuracy results for malware classification models (B,BB,BBA) and meta-model.

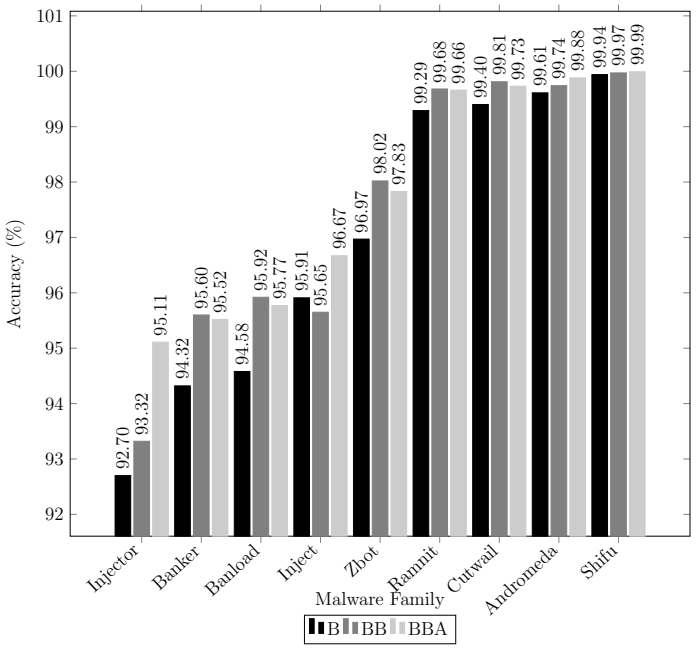

Figure 4.4: Accuracy results for malware families predicted using our malware classification models (B,BB,BBA).

Table 4.2: Precision and Recall Results

Metric	Model	Andromeda	Banker	Banload	Cutwail	Inject	Injector	Ramnit	Shifu	Zbot
Recall	B	99.63	84.56	78.80	90.26	92.39	71.56	97.95	99.77	90.49
	BB	99.29	90.10	85.14	97.16	93.08	72.81	99.13	99.34	93.75
	BBA	99.89	85.21	84.95	95.17	93.67	81.50	99.09	99.56	91.46
Precision	B	91.44	76.92	84.76	92.58	79.53	85.79	97.35	97.34	89.38
	BB	94.61	79.79	87.07	97.29	79.33	92.19	98.78	99.34	93.11
	BBA	97.32	85.54	85.44	96.96	83.59	87.46	98.64	99.78	94.45

of the Banload, Cutwail, and Ramnit families. The accuracy, precision, and recall results indicate that the classifier based on bytes and basic features (BB) achieves excellent results. For instance, the precision results show that given all the samples that were labeled as Cutwail, our BB model was correct 97.29% of the time, outperforming the results obtained for the other two models (92.58% and 96.96% for B and BBA, respectively).

On the other hand, our classifiers yielded average results for the Banker and Banload families. This phenomenon can be explained from the connection between these two malware classes, as Banload is a family of trojans that includes code to download other malware, usually members of the Banker family [55]. As a result, this connection might have hindered the ability of our classifiers to distinguish between these two malware families and make accurate predictions. Similarities in the malicious code of samples for both Inject and Injector families can also explain the modest outcomes generated by our classification models.

For our meta-model, we ran two configurations: a single layer perceptron and a deep learning model similar to the configuration used for our three malware classification models. This allowed us to quantify the improvements that could be obtained by

using deep neural networks in contrast to simpler neural network architectures. Our meta-model, trained using a deep neural network, was able to achieve a much higher accuracy (90.42%) compared to a simple perceptron model (which yielded an average accuracy of 83.22%) to select the simplest model to classify the samples in our dataset.

4.4 Discussion

4.4.1 Meta-Model

The results for our meta-model show that by using bytes features only, we can determine the simplest classifier to correctly assign malware into their corresponding families. Therefore, our meta-model guarantees a speed up for the problem of predicting malware families, as it estimates when less expensive malware characterization, such as byte features, will suffice to accurately classify malware. Our meta-model also correctly predicts the small fraction of malware that require computationally expensive static analysis, such as basic and assembly features. This enables us to decrease the overall time it takes to generate features for a large malware corpus. By using our meta-model to predict the simplest features and models that work best for different malware datasets, we can scale static analysis to upwards of 10-100 million malware samples, as we will only have to use the most expensive static characterizations for a small fraction of the malware in the dataset.

4.4.2 Time Savings

The time required for bytes analysis of a malware is on average 0.06 seconds and is therefore the fastest of our static analysis techniques, and also explains why we

have chosen this feature set to train our meta-model.

The meta-model's time cost is computed by taking the predictions made across each fold by the meta-model, finding the associated cost of the predicted model (B, BB, BBA) for each malware sample in the fold, and computing an average over all samples. Although the meta-model shows slightly higher time cost (7.23 seconds) than the BB model (4.12 seconds), it provides drastically reduced time cost in contrast to the BBA model (which takes over 60 seconds to construct the features for that model). In summary, our meta-model can achieve an accuracy similar to the BBA model, with a running time similar to the BB model.

4.5 Related Work

Furthermore, there are a few related works in the area of neural networks applied to malware classification, based on different characterizations of malicious executables. Dahl et al. [23] used random projections to reduce the dimensionality of malware binary features. Using this reduced input space of strings, and API tri-grams extracted from their malicious samples, they tested multiple neural network architectures, achieving an error rate of 0.49% for a single neural network configuration.

Huang and Stokes [33] introduced a comparison between shallow neural networks and deep learning architectures for the problem of malware classification. Focusing on multi-task learning rates, the authors demonstrated that by employing a two-hidden layer configuration, it was possible to get modest improvements compared to a single hidden layer architecture, and achieved error rates of over 2.94% for the classification

of malware.

A study similar to the work presented in this paper was published by Ahmadi et al. [2]. In their research, features such as metadata, byte-entropy histograms, byte-sequences, number of api calls, and string length were extracted from the hex view and assembly view of a dataset of approximately 20,000 malware. Using XGBoost, a parallel implementation of the gradient boosting tree classifier, the authors managed to select relevant features to classify malware, with an overall accuracy of 99.8%.

In addition, we improve upon the results introduced by Ahmadi et al., because our meta-model can determine the simplest features and models that work best for different unseen malware datasets. As shown in our work, cheap and fast static characterization of malware, such as bytes features, holds enough predictive power to accurately classify malware, and more expensive characterizations such as basic and assembly features are not always required. Furthermore, some families might only need a specific type of features to be correctly classified (e.g., bytes and basic features for cutwail, as shown in Section 4.3). By leveraging our meta-model, we can generate the most expensive static characterizations only for a small subset of a large malware corpus, allowing us to scale static analysis to upwards of 100 million malware samples.

Chapter 5

FEATURE AND MODEL SEARCH

In this chapter, we will describe experiments intended to refine our deep learning models for the malware family classification problem. One of the most important parts of training deep learning models, particularly deep neural networks, is finding the best model configuration and feature set combination. The following sections will discuss the results of our search respectively.

5.1 Model Search

Finding the most accurate model for detecting and classifying malware is our goal. The number of models that need to be trained and validated are a function of the number of hidden layers, their sizes, and other variables such as learning rates. Obtaining a model that provides robust and accurate predictions depends on the input features and the model's configuration. In this section, we use supercomputing resources to achieve an optimal deep learning model.

5.1.1 Titan Supercomputer

Titan [48] is a supercomputer built by Cray at Oak Ridge National Laboratory for use in a variety of science projects. It is comprised of both CPUs and GPUs and is the first such hybrid to perform over 10 petaFLOPS. In this experiment, we need

to train a huge amount of varying models to compare their effectiveness at classifying malware samples. In our previous experiments, we used an AWS framework to complete the training of our models. However, given the exreme scale of our search space, we require huge compute capabilities that only a supercomputer can provide. We achieve a great speedup in training time using Titan as we can train all models simultaneously.

Titan contains 18,688 physical compute nodes, each with a processor and physical memory. Each of these compute nodes contains a 16 core processor with 32 GB of RAM. Spider, Titan's extremely high-performance file system, has over 26,000 clients, providing 32 petabytes of disk space and can move data at more than 1 TB/s. As we will detail in the next section, by using these resources we can train and validate all 791 deep learning models in parallel using 248 compute nodes while storing their model data and results instantaneously across sessions.

5.1.2 Experiment and Results

In this section, we will detail our initial Titan model search experiments. In the field of malware detection and classification there is minimal effort in choosing the "best" model configuration or model hyperparameters. In most cases, a few model designs are selected and tested and the best is chosen from that select group. This experiment is an exhaustive search of model configurations to find the optimal deep learning model for classifying malware samples into families.

5.1.2.1 Model Configurations

Discovering the best model configuration for a neural network is a difficult task as most researchers rely on previous domain knowledge or expert systems to design their models. For this experiment, we choose our search to contain a number of hidden layers from one to five and the hidden layer sizes range from 8 to 512 by powers of two. Previous experiments have shown that more than 5 hidden layers have not produced better results than smaller models. Also, hidden layer sizes below 8 were not ideal as the number of targets, or number of malware families, we are trying to classify is 8. The only other stipulation on model configuration is that the hidden layer sizes must be decreasing from the first hidden layer. The number of configurations per hidden hidden layer can be seen in Table 5.1.

Table 5.1: Number of Configurations per Number of Hidden Layers

Hidden Layer Size	Count
One	7
Two	28
Three	84
Four	210
Five	462
Total	791

5.1.2.2 Dataset

Similarly to the dataset we described in 4.2.3, we obtained our malware samples from Reversing Labs and selected a subset to use for our model search. Contrary to the dataset in 4.2.3, we removed confounding malware families that are extremely similar in their characteristics and behaviors to get a more accurate prediction.

Initially, we optimized the size of our dataset to the number of parallel models we could train on a single Titan core. By using a dataset on the scale of ten thousand, we were able to make use of every available CPU on each core giving us 100% utilization of our requested nodes. Using a much larger dataset on the scale of one hundred thousand, the 32 GB of RAM per node described in Section 5.1.1 would not be enough to fully utilize each nodes compute capability. The breakdown of malware families can be seen in Table 5.2. We choose to include Goodware in our dataset as our model's can be considered both a malware detector and malware classification model.

Table 5.2: Composition of Malware and Goodware Dataset

Name	Type	Count
Andromeda	virus	1250
Banker	spyware	1250
Cutwail	downloader	1250
Goodware	benign	1250
Inject	virus	1250
Ramnit	trojan	1250
Shifu	spyware	1250
Zbot	downloader	1250
Total		10,000

5.1.2.3 Results

The exhaustive model search was computed on Titan by 5-fold cross validation using the dataset disucssed in the previous section. All 791 models and their corresponding error rates sorted by hidden layer size can be seen in Figure 5.2. From this figure, we can see that there exists a tight grouping of error rates for sizes one through three, while sizes four and five have a much greater dispersion. This dispersion is most likely due to the sheer number of models generated by the exhaustive search. We can

also see that every model with hidden layer sizes of two and three contain smaller error rates than every model with one hidden layer. A closer look at models with less than five hidden layers can be seen in Figure 5.3

The number of hidden layers that achieves the lowest error rate in the search is four, but there is a great dispersion of error rates for models of that size. Models with five hidden layers have an even greater dispersion, with most models being worse than smaller models. Furthermore, these large models take significantly longer to train than their smaller counterparts. The top ten model configurations can be seen in Table 5.3. The most accurate model configuration's confusion matrix can be seen in Figure 5.1. As we saw in Section 3.2 and confirmed in this experiment, inject and zbot are difficult malware families to classify.

Table 5.3: Top Ten Model Configurations

# of Hidden Layers	Model Configuration	Error Rate
4	256-128-16-16	5.05%
4	256-128-64-32	5.08%
3	512-256-128	5.09%
3	512-32-8	5.1%
4	256-128-64-64	5.11%
4	512-512-256-128	5.12%
3	256-32-32	5.12%
3	256-128-32	5.12%
4	128-128-128-16	5.12%
5	256-256-128-64-16	5.13%

Every model evaluated in the model search can be seen in Figure 5.4. This figure shows the error rate of the epoch that achieved the highest accuracy for all 791 models. From the figure, we can deduce that most models that attain their best accuracy before epoch 140 do not achieve a low error rate. Models that continue to learn past epoch

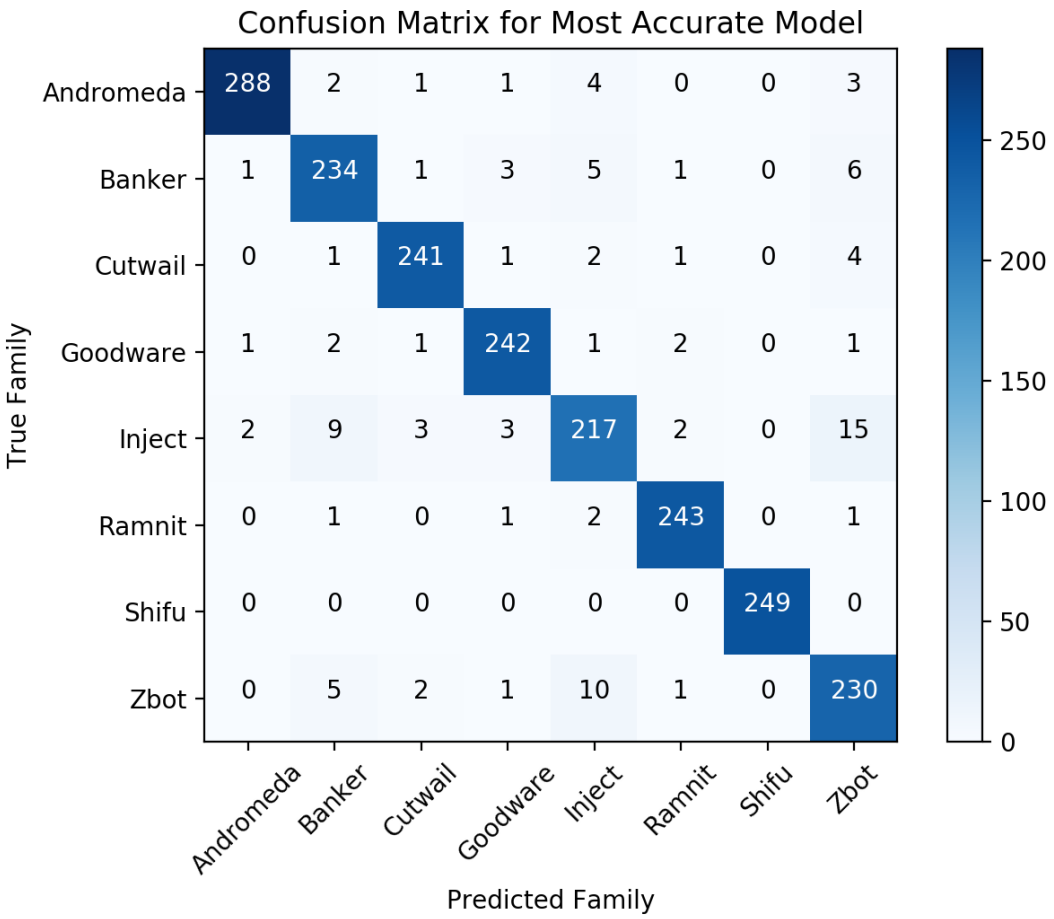

Figure 5.1: This figure shows the confusion matrix for the top performing model out of all 791 models tested.

140 tend to outperform most models. This information would be essential to know in a non-exhaustive model search or a feature search. Models that do not decrease their error rates significantly before 140 epochs could be discarded early.

The time to train a model is a significant factor in determining the optimal model configuration. Figure 5.5, shows a simple breakdown of model training times per epoch which are split by their respective hidden layer sizes. Due to parallelization in the neural network training code, the average time to train per epoch is relatively the

same across all model configurations. However, the maximum training times seem to increase with hidden layer size which most likely denotes a problem with parallelization due to 100% CPU utilization on the supercomputing node. The large minimum training time for one hidden layer models is most likely due to the small search space as there are only seven models with that configuration. Figure 5.6 shows a more in-depth breakdown of the model's training times per epoch in a box-and-whisker plot.

Further experimentation is completed in the next sections based on the output of this model search. In the next section, we discuss experiments to check the robustness of our exhaustive model search. In Section 5.2.2.1, we discuss using a Genetic Algorithm to search for the optimal subset of all features. We have chosen the simplest model in the top 5 performing models from this experiment as we will be using less features to generate highly accurate models.

5.1.3 Further Experimentation

Selecting the best model configuration depends on the targets of the model, the input features used, and in some instances the dataset used to train the model. In this section, we posit that our model search does not depend on the specific dataset used to train the models and the optimal models will remain optimal given a different dataset.

In this experiment, we increase the number of malware families slightly and the size of the dataset tenfold. In order to test the effectiveness of our model search, we select ten top performers, middle performers, and bad performers from the model configurations based on their error rates on the smaller dataset. The pipeline for our

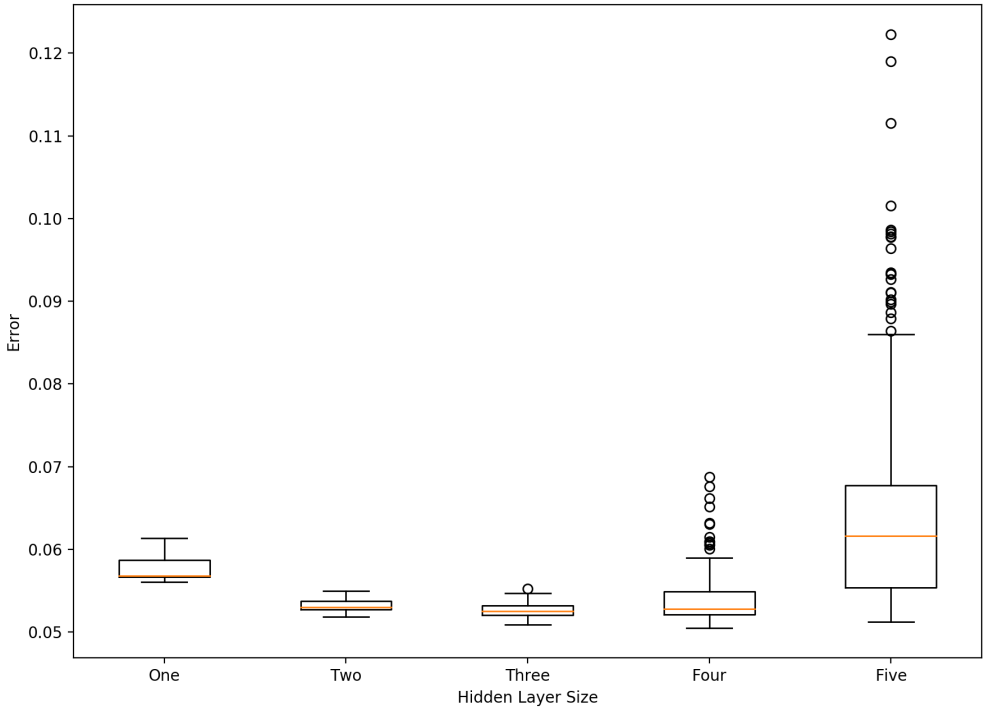

Figure 5.2: This figure shows the results of our exhaustive model search run on the Titan supercomputer separated by the hidden layer size of the model. Due to the sheer number of four and five hidden layer models, there is a greater dispersion among error rates than smaller model configurations. One hidden layer is not enough to obtain a good error rate as every model with two and three hidden layers performs better.

scaled up experiment can be seen in Figure 5.7.

5.1.3.1 Dataset

For this section, we designed an experiment to test the results of our exhaustive model search on a new, much larger dataset. In Section 4.2.3, we described a dataset that is ten times larger than the one used in the previous section. The breakdown of the malware samples families and counts can be seen in Table 4.1. Using this dataset, our new experiments will provide us with confirmation that the leading models selected

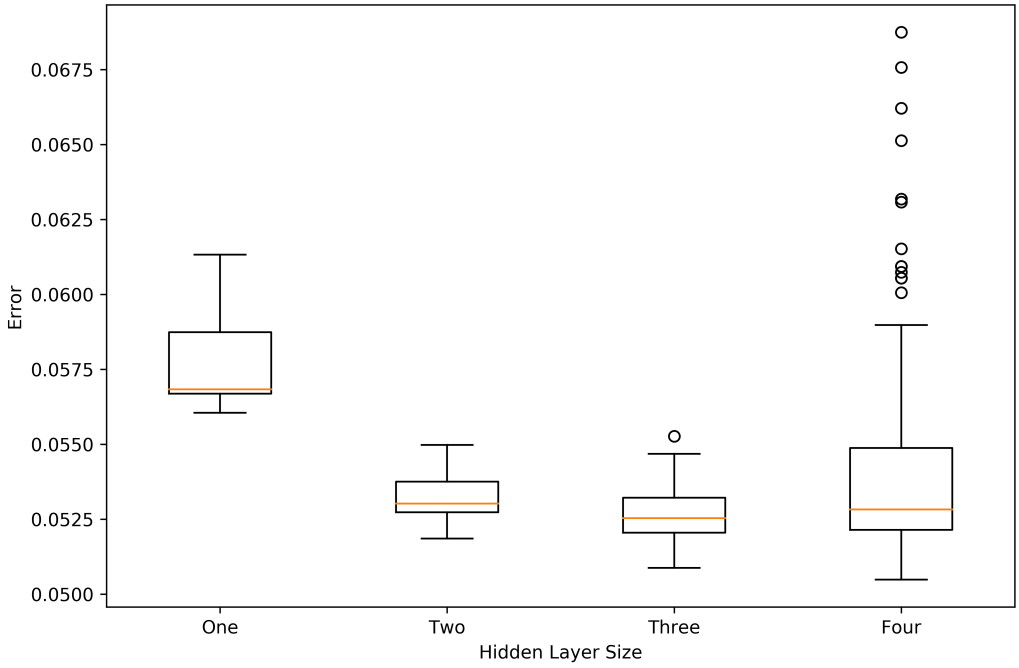

Figure 5.3: This figure shows a closer look at the hidden layer sizes from one to four. The lowest error rate is found in a model configuration with four hidden layers.

from the model search are robust to changes in dataset composition and size.

5.1.3.2 Results

The results of this experiment provide further evidence that given a static feature set, top model performers tend to remain top performers with seperate datasets. In Figure 5.8, you can see the results of the model configurations on a new dataset. The original sections (top, middle and bottom model performers) were tested with a new dataset to determine if they remain in the same section of performance when ranked.

70% of the top model performers from the exhaustive search are ranked in the top performers in the new experiment. This result provides us with a good indication

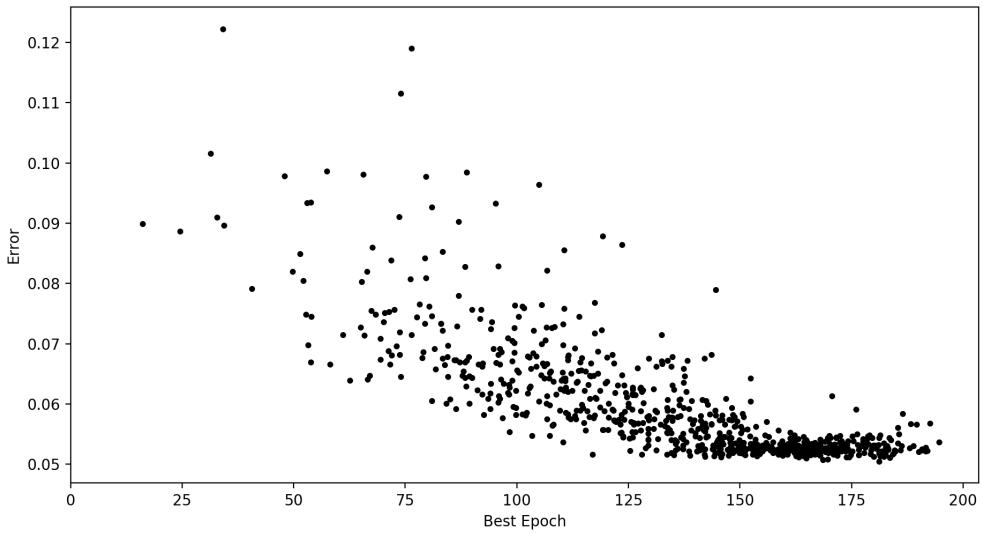

Figure 5.4: This figure shows the best epoch resulting in the lowest error rate for every model in the exhaustive search. Between epochs 140 and 200 there exhibits a tight grouping of high accuracy models. In general, models that have best epochs below 140 tend to not learn as well and therefore exhibit a lower accuracy.

that with a static feature set, optimal model configurations tend to remain good performers for malware detection and malware family classification given a new dataset. The top five performing models of the new experiment can be seen with their original rankings in Table 5.4.

However, the middle and bottom ranked results from the exhaustive search did not contain the same result as the top performers on a new dataset. In fact, the worst performers from the model search tended to perform better than the middle models. This may be a case in which both middle and bottom performing models in the model search are simply bad performing models and that the sheer number of models produced in the exhaustive model search allowed them to be ranked higher than other outliers. Combining the middle and bottom performance sections together,

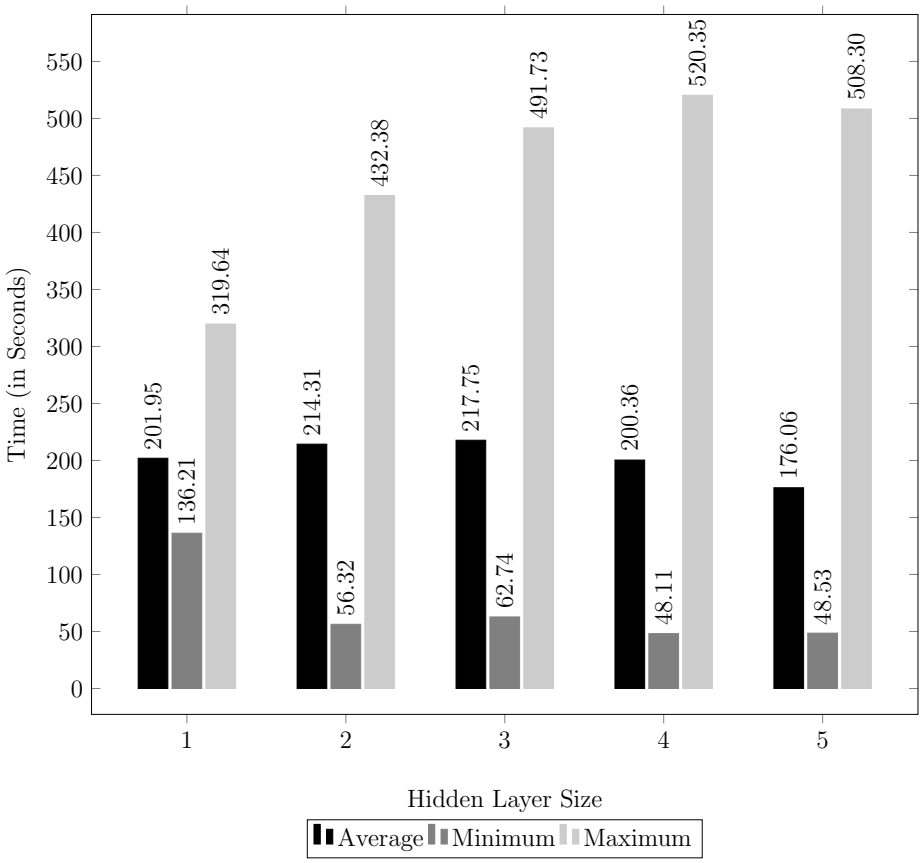

Figure 5.5: This figure shows the average, minimum, and maximum training times per epoch for all models of a given hidden layer size. On average, each hidden layers models take around 200 seconds per epoch to train. As the dataset is relatively small, most of the RAM can be used for parallelization in the hidden layer training code which allows them to be trained almost simultaneously. The maximum training times are increasing with hidden layer size because in worst-case there is no parallelization and training takes longer with a larger model.

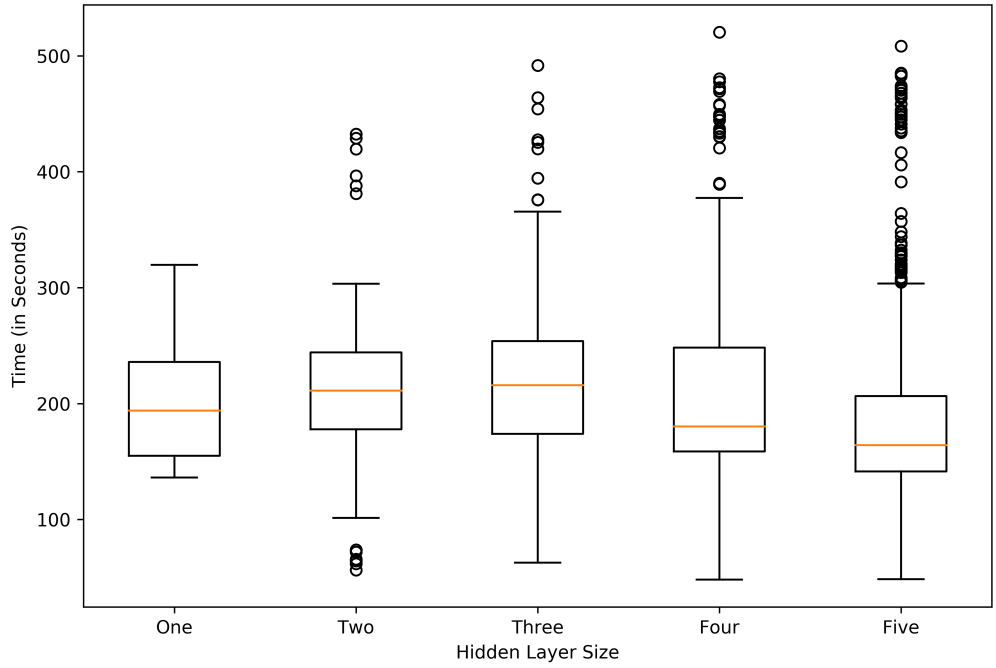

Figure 5.6: This figure depicts a box and whisker plot for all average training times within models. We can clearly see that models of four and five hidden layers contain a large number of statistical outliers. An interesting insight is that models of two hidden layers are the only models that contain outliers that train very fast which could be a result of good parallelization.

85% of the models are correctly placed which is added evidence that both middle and bottom performance sections are equally bad performing model configurations.

5.1.4 Related Work

Finding the best model for a specific application is a very difficult problem to complete fully and with great certainty. Some applications of malware detection such as Menahem et al. [54, pp. 1483-149] get around searching for the best base classifier and choose to create ensemble methods to combine classifiers into more accurate and

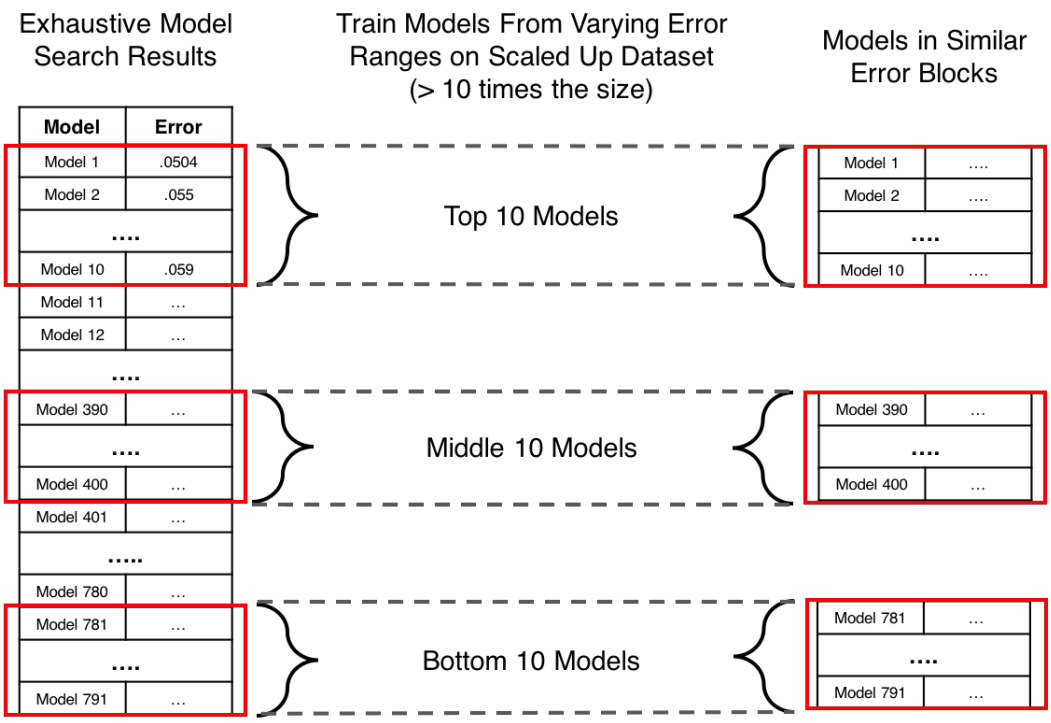

Figure 5.7: This figure shows the pipeline for further experiments with our exhaustive model search results. This experiment to to see whether or not the exhaustive model search results depend on the specific dataset or are generalizable for any of our malware sample datasets based on using every feature available. We take 10 models from the top, middle and bottom performers of the model search and train them using 5-fold cross validation on a much larger, scaled up dataset with more malware families. If the results stay in similar groups of performance, we can say with confidence that the model search results are generalizable.

robust models.

Within deep learning, there needs to be investigation in the correct model configuration and hyperparameters. Grosse et al. [30] completes this task by creating models with layer sizes from one to four and varying hidden layer sizes from 10 to 300 while the hyperparameters were chosen based on previous knowledge. This kind of exhaustive search is similar to our search, but on a much smaller scale. Within a

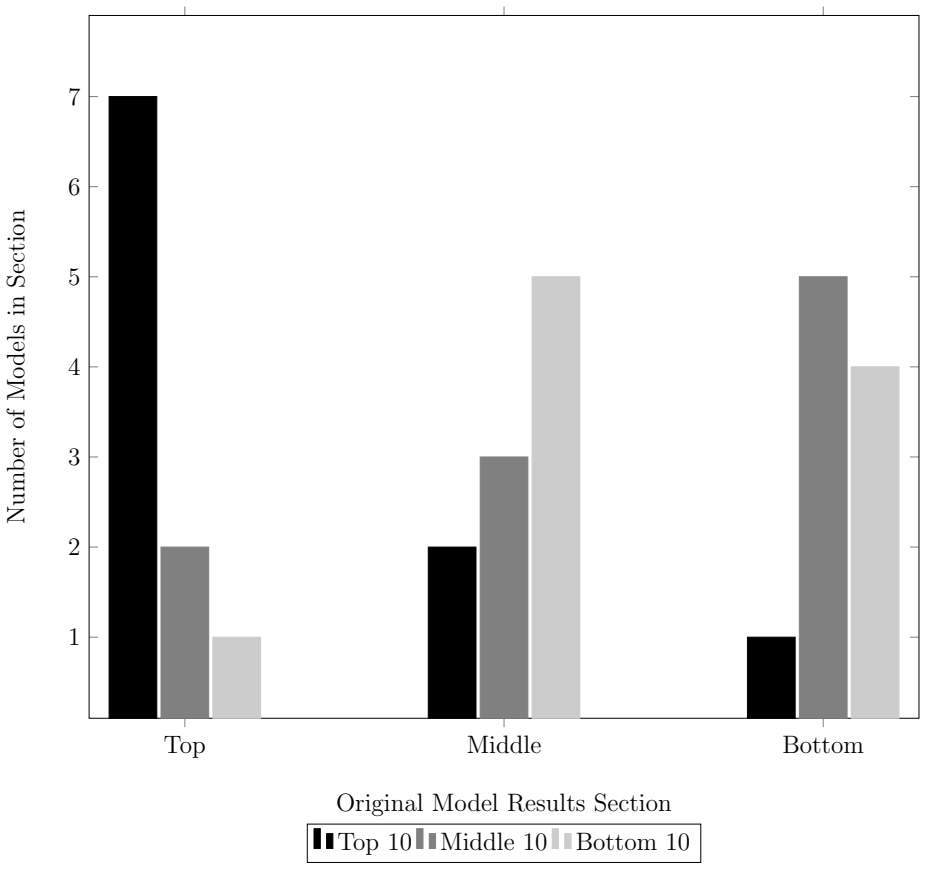

Figure 5.8: This figure shows the results of our new experiment to test the models from our model search experiment on a new, much larger dataset. The results for the middle and bottom tier models are mixed. However, most top tier models from the original model search appear in the top performing models of this search.

Table 5.4: Top Five Model Configurations

Original Result	Model Configuration	Error Rate
7	256-32-32	9.38%
4	512-32-8	9.44%
8	256-128-32	10.5%
3	512-256-128	11.5%
2	256-128-64-32	11.7%

different field of research Potok et al. [64, pp. 47-55], uses TITAN supercomputer and evolutionary optimization to search for the optimal deep learning topology given the MNIST handwriting dataset.

5.2 Feature Search

Feature set exploration is computationally intensive, but a very important part of deep learning. In Chapter 4, we discuss the varying costs of our feature sets and how we can create a low cost and highly accurate model. In this section, we focus on what are the most important features to use as input to produce highly accurate models. We start by discussing exploratory preliminary work which shows the varying predictability of singular features. Then, we will discuss more advanced techniques in finding optimal feature sets using genetic algorithms.

5.2.1 Preliminary Work: Single Feature Models

A natural starting point for finding the top performing features is to build models with a single feature as input. Using all bytes, basic and assembly features and every possible combination of normalization methods that were applicable to those features we obtain 47 distinct features we will use as inputs. All 47 features, their

categories, and normalization methods can be seen in Table 5.5. The single feature models were built with our deep learning framework we described in Section 2.4.4.

The dataset we used for this experiment is over 100,000 malware samples and was introduced in Section 4.2.3. The model architecture we used was a simple perceptron model which should give a fair estimate of the features prediction power. The results of the experiment can be seen in Figure 5.9 and are ordered from most accurate to least accurate.

As the results indicate, single feature models are not robust enough to accurately predict a malware's family even though most models are better than a random guess (11.11% accuracy with nine families). However, the results do show a pattern of the best features to use in our malware family classification models. The top ten performers used byte or basic feature categories. Specifically, the best basic analysis performers were features obtained from extracting PE Header information. This is an interesting conclusion as byte and basic features are the least costly to produce, but seem to have tremendous predictive power.

5.2.2 Genetic Algorithms

A genetic algorithm is a heuristic search inspired by Charles Darwin's theory of natural evolution. Genetic algorithms are used widely to produce optimal solutions to problems in many fields. The key components in a genetic algorithm include:

- A population of individuals
- Fitness function

- Selection Method
- Crossover (or recombination)
- Mutation

The biggest challenge in creating a genetic algorithm is translating the problem into a chromosome that represents an individual. A chromosome is made up of a set number of genes and the initial population is created by constructing a random chromosome for each individual. A fitness function is used to evaluate the individual so that the population can be ranked so that selection can occur. Selection is usually completed by taking the top performers of the population and potentially some low performers and selecting them as breeders. These breeders perform crossover by exchanging part of their chromosomes to create children. Each child created has a small chance of their chromosome mutating or changing slighly. One pass through this process is called a generation and in most cases, many generations must be completed before an optimal solution is found.

5.2.2.1 Feature Search

In this section, we will describe the feature search we implement using genetic algorithms. We select a neural network model with three hidden layers of size 512, 32, and 8. This model configuration performed well under both experiments performed with two seperate datasets described in Section 5.1. The dataset we use is described in Section 5.1.2.2 and consists of eight malware families with a combined 10,000 malware

samples. We use a small dataset for our initial test as the genetic algorithm normally takes many generations to converge to an optimal solution. The input of model is determined by the chromosome of the individual which we will describe below.

In our genetic algorithm, a gene is an individual feature. The chromosome is made up of all 47 available features, or genes, represented by a 0 or a 1 if the feature does not exist or exists in the model respectively. Each individual can then be represented by a binary array of length 47 which further represents a subset of features which will be used as input for our neural network. In Figure 5.10, a depiction of the chromosome and an example of an individual gene is shown.

Each individual is trained and tested using 5-fold cross-validation for which an error rate is computed. Each model is trained 50 epochs, which is enough to accurately evaluate the feature set's predictability. The fitness function for this genetic algorithm is the error rate of the model. A population consists of 20 individuals. The top 6 performers are selected as breeders and 2 random "lucky" individuals are chosen from the remaining 14. These breeders are selected randomly to exchange information and create children by selecting a single crossover point to create two new chromosomes. The process of crossover can be seen in Figure 5.11. Then, each gene in each chromosome in the population has a mutation rate of 5%. This means that there is a 5% chance of mutation for the each bit to change from a 0 to a 1 or vice versa. The full genetic algorithm pipeline for this application can be seen in Figure 5.12.

5.2.2.2 Results

The goal of the genetic algorithm should be to find the best set of input features given a specific deep learning model configuration. In each generation, the best model is selected based on the minimum error rate amongst all individuals. The difference between the current generation's minimum error model and the last will become our heuristic for stopping criteria. When the difference becomes close to 0, we assert that the genetic algorithm has found the optimal feature set. We also record the average error rate amongst all individuals within a generation. This metric will provide us with more validation that our stopping criteria is correct and our local minimum is in fact global.

In this experiment, we find the fittest individual, and therefore optimal feature set, in generation 43 with an error rate of 7.04%. The minimum error difference becomes zero at generation 42, which posits that the individuals are close to a local or global minimum in terms of error rate. We continue evaluating the genetic algorithm almost four times as long as this minimum to allow for other local minimum's to be seen. Zero minimum difference is not seen again after 160 generations and no other feature set attains a smaller minimum error. With this in mind, we deduce that the best feature set within generation 43 is a global minimum.

A graph of the individual with the minimum error rate in each generation can be seen in Figure 5.13. We also look at the average error rates of all individuals within each generation. In Figure 5.14, we can see that generation 43 is very close to a global minimum for average error which occurs in generation 38. We can infer that the

generations around this global minimum will contain the fittest individuals within our entire search.

The fittest individual's genetic composition can be seen in Figure 5.6. The feature number corresponds to the numbers in Table 5.5 and to a value of 1 in the individual's chromosome at that index. The optimal feature set uses only 21 out of the potential 47 features. This is a drastic reduction of the feature space, which reduces the cost of generating such features during analysis and the time cost of training and validating our deep learning models. We continue improving the results of this experiment in the next section.

5.2.2.3 Further Analysis

The top overall performer from the best generation was chosen to be trained further. Initially, we train the models 50 epochs to decide whether the feature set is learning efficiently. We train the overall performer to 200 epochs and select the optimal error rate for that model. In doing this, we obtained a 5.41% error rate which is a highly accurate model for this dataset.

This model contains only 21 out of the possible 47 available features and is almost as accurate as our best model in Section 5.1.2.3 which contains all 47 features. The genetic algorithm's best model is not as accurate as our best model for this dataset, but it does provide a lower-cost and faster training model. As most of the features that were chosen were byte features, we use most of our low-cost and fast to produce features while selecting the best higher cost features from the basic and as categories.

5.2.3 Related Work

Feature selection is a large part of making accurate and efficient machine learning models. Baldangombo et al. [7] uses information gain to select the best PE header information for malware detection. Raman et al. [67] experimented with different feature selection methods on PE header features to find a minimal set of features that obtained a reasonable detection rate. Dahl et al. [23, pp. 3422-3426] collects data from the malware's process memory and API calls. Feature reduction is then completed using mutual information and then random projections to further reduce the feature space. Using behavioral data, Lin et al. [49, pp. 965-992] uses TF-IDF (term frequency-inverse document frequency) and PCA to reduce the dimentionality of the feature space for malware classification.

In android malware detection, feature selection and reduction methods have been extensively surveyed by Pehlivan et al. [63]. Ranveer et al. [68] compared feature extraction methods used for malware detection and surveyed strengths and weaknesses of the current literature in all malware detection platforms.

Using neural networks, Jiang et al. [40, pp. 890-895] also uses PE header information, but creates their own feature selection algorithm based on information gain to reduce their features for malware detection. Some researchers choose to use combinations of feature selection methods. Cepeda et al. [15, pp. 560-566] uses chi-squared feature reduction in addtion to Mean Decrease in Accuracy (MDA) and Mean Decrease in Impurity (MDI) feature selection methods to reduce the dimensions of the Virustotal data which is used for malware detection.

Table 5.5: All Available Features

Feature #	Feature Category	Feature	Normalization
1	bytes	metrics	ID
2	bytes	metrics	logcount
3	bytes	histogram	ID
4	bytes	histogram	logcount
5	bytes	histogram	freq
6	bytes	byte-entropy histogram	ID
7	bytes	byte-entropy histogram	logcount
8	bytes	byte-entropy histogram	normalize
9	bytes	byte-entropy histogram	normalize-bytes
10	bytes	byte-entropy histogram	normalize-entropy
11	basic	metrics	ID
12	basic	metrics	logcount
13	basic	strings	ID
14	basic	strings	logcount
15	basic	strings	normalize
16	basic	metadata	ID
17	basic	metadata	logcount
18	basic	metadata	normalize
19	basic	import	ID
20	basic	import	logcount
21	basic	import	normalize
22	assembly	stats	ID
23	assembly	stats	logcount
24	assembly	1grams	ID
25	assembly	1grams	logcount
26	assembly	1grams	freq
27	assembly-spectrums	functions-eigenvals	ID
28	assembly-spectrums	functions-stats	ID
29	assembly-spectrums	functions-stats	logcount
30	assembly-spectrums	functions-stats	normalize
31	assembly-spectrums	functions-1grams	ID
32	assembly-spectrums	functions-1grams	logcount
33	assembly-spectrums	functions-1grams	normalize
34	assembly-spectrums	blocks-eigenvals	ID
35	assembly-spectrums	blocks-stats	ID
36	assembly-spectrums	blocks-stats	logcount
37	assembly-spectrums	blocks-stats	normalize
38	assembly-spectrums	blocks-1grams	ID
39	assembly-spectrums	blocks-1grams	logcount
40	assembly-spectrums	blocks-1grams	normalize
41	assembly-spectrums	operations-eigenvals	ID
42	assembly-spectrums	operations-stats	ID
43	assembly-spectrums	operations-stats	logcount
44	assembly-spectrums	operations-stats	normalize
45	assembly-spectrums	operations-1grams	ID
46	assembly-spectrums	operations-1grams	logcount
47	assembly-spectrums	operations-1grams	normalize

Figure 5.9: This figure displays results of our single feature perceptron models. All 47 feature and normalization method combinations are displayed with their corresponding error rates.

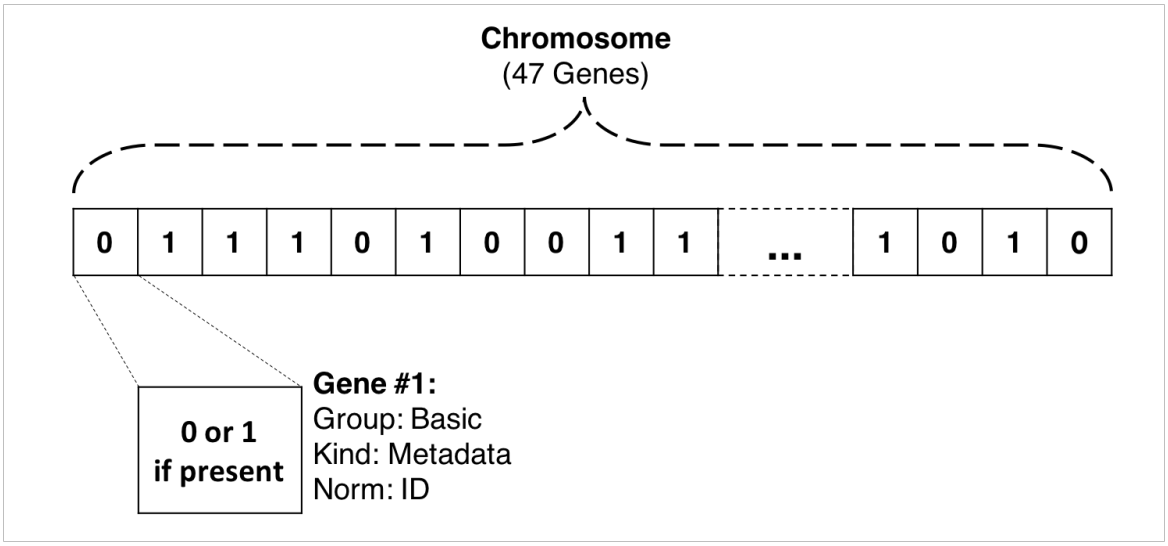

Figure 5.10: This figure describes the definition of a chromosome within our genetic algorithm. Each gene represents a single feature where a 1 is that the feature is an input to the model and 0 is the opposite. There are 47 genes which make up our chromosome which fully expresses the input features of the model.

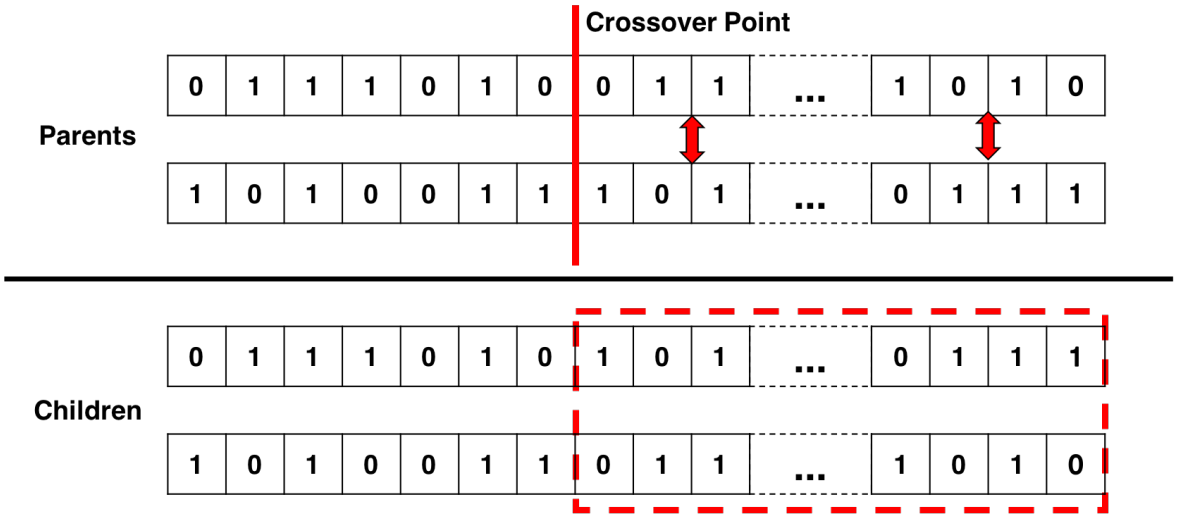

Figure 5.11: This figure describes the process of crossover within our genetic algorithm. The two parents are chosen from the available breeders and a random crossover point is selected. All genes to the right of the crossover point are swapped between the two chromosomes. The resulting two chromosomes are the resulting children and begin to fill the next generation's population. The mutation process happens after crossover is completed.

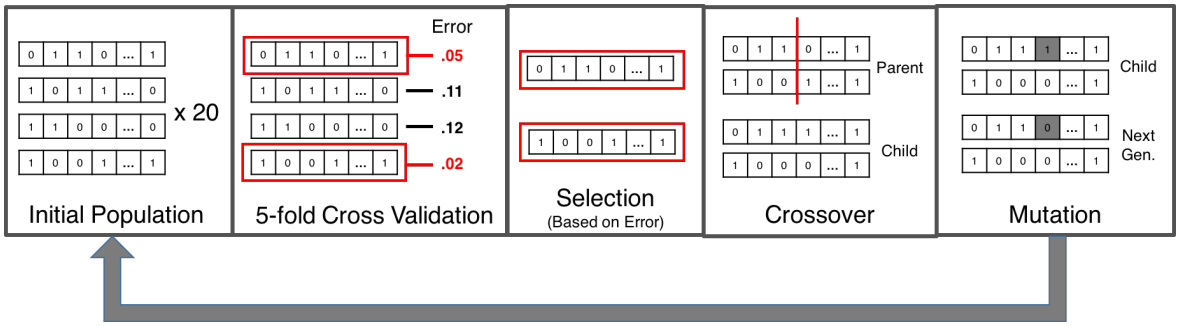

Figure 5.12: This figure describes the full pipeline of the genetic algorithm. First, a population of 20 individuals are created randomly to start the process. Then, each individual is evaluated using 5-fold cross validation. The resulting error values are used to select the fittest individuals and a few lucky individuals to be parents. Crossover is completed and a new set of children are created. These children go through random mutations of their chromosomes and become the new generations initial population. This pipeline recurses until an optimal answer is acheived.

Table 5.6: Optimal Feature Set Using Fittest Individual

Feature #	Feature Category	Feature	Normalization
1	bytes	metrics	ID
2	bytes	metrics	logcount
3	bytes	histogram	ID
4	bytes	histogram	logcount
5	bytes	histogram	freq
6	bytes	byte-entropy histogram	ID
7	bytes	byte-entropy histogram	logcount
9	bytes	byte-entropy histogram	normalize-bytes
11	basic	metrics	ID
16	basic	metadata	ID
17	basic	metadata	logcount
20	basic	import	logcount
22	assembly	stats	ID
23	assembly	stats	logcount
25	assembly	1grams	logcount
27	assembly-spectrums	functions-eigenvals	ID
29	assembly-spectrums	functions-stats	logcount
30	assembly-spectrums	functions-stats	normalize
33	assembly-spectrums	functions-1grams	normalize
40	assembly-spectrums	blocks-1grams	normalize
45	assembly-spectrums	operations-1grams	ID

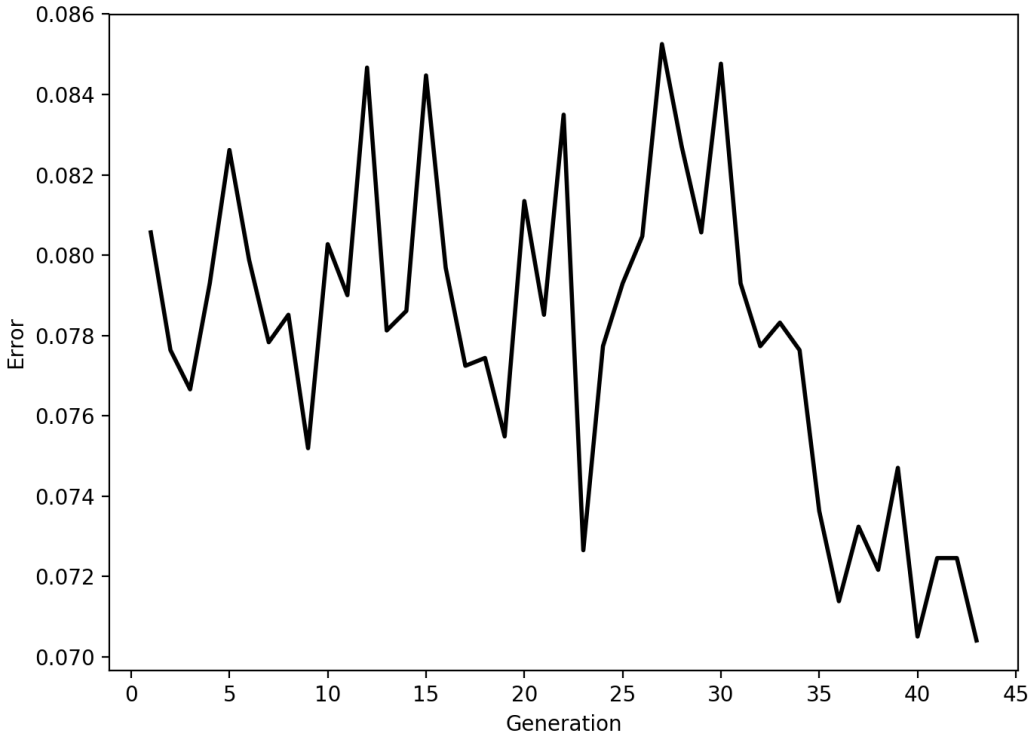

Figure 5.13: This figure shows each generation's fittest individual chosen by error rate. The error rate declines until a minimum error is seen. The algorithm reaches other local minimum, but this graph shows the global minimum.

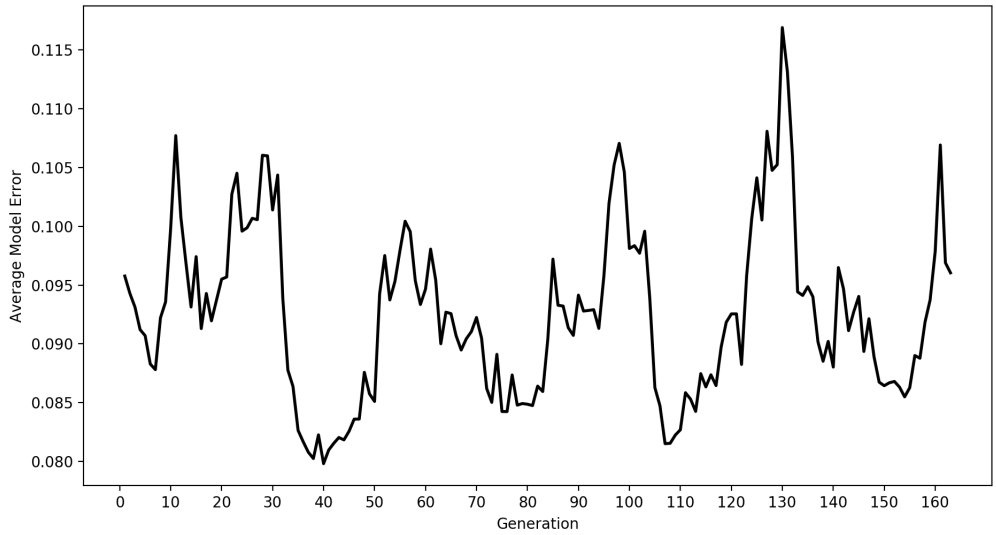

Figure 5.14: This figure shows the average error of all individuals in each generation. We use this graph to obtain more evidence that our fittest individual is in fact a global minimum. We can see that the global minimum for average error is close to our minimum error feature set. Generations around this global minimum will contain the fittest individuals which infers that our optimal feature set is a global minimum.

Chapter 6

CONCLUSION AND FUTURE WORK

6.1 Conclusion

In this dissertation, we have described using machine learning for the detection and classification of malware. Using static analysis of malware we can generate descriptive features to be used in conjunction with deep supervised learning models in order to predict a file's maliciousness. Our contribution to the research fields of malware detection and malware family classification includes: feature set and model exploration, exploring the tradeoffs in dataset size, accuracy of malware detection and classification models, and the cost of producing large-scale datasets and deep learning models.

This work advances the state-of-the-art in the fields of malware detection and malware family classification. Malware datasets in the current literature are far too small to create realistic models for the detection or classification of malware. Reducing the cost of generating large malware datasets will allow researchers to more easily create realistic datasets to train on. Using our meta-model design, security analysts can make less costly, faster, and more accurate malware detection and malware family classification predictions. We have discovered the best static features to be used in both malware detection and malware family classification models so that we can increase

the accuracy of our models and reduce the time it takes to generate static features. In addition, we have explored the best model configurations to use given basic, byte and assembly features and we have documented the optimal architectures. We hope that further research in this area provides us with even faster and more accurate malware detection models to secure any organization.

6.2 Future Work

This research can be viewed as the start of cost efficient large scale machine learning for malware detection and malware family classification. There is much more work to be completed to expand on these experiments and to further validated them. Some ideas for future work are presented below.

6.2.1 Datasets

Much of the work presented in this dissertation focuses on how to build large scale machine learning models while keeping monetary and time cost low. The largest dataset we build for use in these experiments is on the scale of one hundred thousand. In a realistic environment, we would want to train our machine learning models on millions of malware samples. These malware samples should be balanced between malware families and should represent as many families as possible. Validating the results of these experiments, specifically the meta-model discussed in Section 4, should be completed with a large-scale realistic dataset.

Some of the data transformation we use are best practices from previous research. Specifically, the basic features we generate are transformed into fixed sized

vectors and this transformation can be completed in many ways. The technique has many factors such as the hashing function and the size of the vector which can be changed to produce a different end result. The predictability of new transformations should be researched.

Time is an important characteristic to understand in malware detection. In our case, we looked at reducing the time it takes from malware analysis to detection. However, an organization faces threats of constant malware attacks. Therefore, time-series analysis of malware detection must be completed. Our models can be validated by testing their effectiveness in a period of time, re-training them with the new malware attacks, and further validating them in the next time period. This will show our models effectiveness to learn from previous attacks.

6.2.2 Feature Search

In Section 5.2.2.1, we discuss a feature search using genetic algorithms to find the best subset of our complete feature set. Our feature search was validated by finding providing a 55% reduction in the feature space while obtaining nearly the same accuracy as our best model configuration using all features. This feature search could be further validated by comparing its effectiveness with different feature reduction technique such as PCA.

The genetic algorithm we developed for our feature search could include genes to represent model configurations as well. This would allow us to test the best feature sets to use with specific models which may generate more accurate models than

searching the model and feature space separately. This would require a large number of generations and would have to be more parallelizable to be able to run in linear time.

6.2.3 Model Search

The exhaustive model search using Titan included searching for the best deep neural network model configuration. Hyperparameters of the model configuration such as the learning rate or the activation function chosen for each hidden layer are not considered and could be important to the overall accuracy of the network. The addition of these hyperparameters would require vast amounts of compute requirements. Given the results of our model search we have uncovered many termination conditions that could be implemented that would reduce the search space dramatically.

BIBLIOGRAPHY

[1] Yousra Aafer, Wenliang Du, and Heng Yin. "Droidapiminer: Mining api-level features for robust malware detection in android." In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2013, pp. 86–103.

[2] Mansour Ahmadi, Dmitry Ulyanov, Stanislav Semenov, Mikhail Trofimov, and Giorgio Giacinto. "Novel feature extraction, selection and fusion for effective malware family classification." In: *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*. ACM. 2016, pp. 183–194.

[3] Faraz Ahmed, Haider Hameed, M Zubair Shafiq, and Muddassar Farooq. "Using spatio-temporal information in API calls with machine learning algorithms for malware detection." In: *Proceedings of the 2nd ACM workshop on Security and artificial intelligence*. ACM. 2009, pp. 55–62.

[4] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. "Graph-based malware detection using dynamic analysis." In: *Journal in computer Virology* 7.4 (2011), pp. 247–258.

[5] Blake Anderson, Curtis Storlie, and Terran Lane. "Improving malware classification: bridging the static/dynamic gap." In: *Proceedings of the 5th ACM workshop on Security and artificial intelligence*. ACM. 2012, pp. 3–14.

[6] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket." In: *Network and Distributed System Security Symposium*. 2014.

[7] Usukhbayar Baldangombo, Nyamjav Jambaljav, and Shi-Jinn Horng. "A static malware detection system using data mining methods." In: *arXiv preprint arXiv:1308.2831* (2013).

[8] Ulrich Bayer, Engin Kirda, and Christopher Kruegel. "Improving the efficiency of dynamic malware analysis." In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. ACM. 2010, pp. 1871–1878.

[9] Zahra Bazrafshan, Hashem Hashemi, Seyed Mehdi Hazrati Fard, and Ali Hamzeh. "A survey on heuristic malware detection techniques." In: *5th Conference on Information and Knowledge Technology (IKT)*. IEEE. 2013, pp. 113–120.

[10] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. "Greedy layer-wise training of deep networks." In: *Advances in neural information processing systems*. 2007, pp. 153–160.

[11] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." In: *Journal of Machine Learning Research* 13 (2012), pp. 281–305.

[12] Jonathan J Blount, Daniel R Tauritz, and Samuel A Mulder. "Adaptive rule-based malware detection employing learning classifier systems: a proof of concept." In: *IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW)*. IEEE. 2011, pp. 110–115.

[13] Avrim L Blum and Pat Langley. "Selection of relevant features and examples in machine learning." In: *Artificial intelligence* 97.1 (1997), pp. 245–271.

[14] Gerardo Canfora, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. "Detecting android malware using sequences of system calls." In: *Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile*. ACM. 2015, pp. 13–20.

[15] Carlos Cepeda, Dan Lo Chia Tien, and Pablo Ordóñez. "Feature selection and improving classification performance for malware detection." In: *Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom), 2016 IEEE International Conferences on*. IEEE. 2016, pp. 560–566.

[16] Silvio Cesare and Yang Xiang. "Malware variant detection using similarity search over sets of control flow graphs." In: *IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE. 2011, pp. 181–189.

[17] Silvio Cesare, Yang Xiang, and Wanlei Zhou. "Malwise—an effective and efficient classification system for packed and polymorphic malware." In: *IEEE Transactions on Computers* 62.6 (2013), pp. 1193–1206.

[18] Girish Chandrashekar and Ferat Sahin. "A survey on feature selection methods." In: *Computers & Electrical Engineering* 40.1 (2014), pp. 16–28.

[19] Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, and Haojin Zhu. "Hardening Malware Detection Systems Against Cyber Maneuvers: An Adversarial Machine Learning Approach." In: *arXiv preprint arXiv:1706.04146* (2017).

[20] Brian Chess and Gary McGraw. "Static analysis for security." In: *IEEE Security & Privacy* 2.6 (2004), pp. 76–79.

[21] Mihai Christodorescu and Somesh Jha. "Testing malware detectors." In: *ACM SIGSOFT Software Engineering Notes* 29.4 (2004), pp. 34–44.

[22] Cuckoo. In: (). URL: https://cuckoosandbox.org.

[23] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. "Large-scale malware classification using random projections and neural networks." In: *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*. IEEE. 2013, pp. 3422–3426.

[24] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. "Yes, Machine Learning Can Be More Secure! A Case Study on Android Malware Detection." In: *IEEE Transactions on Dependable and Secure Computing* (2017).

[25] Tobias Domhan, Jost Tobias Springenberg, and Frank Hutter. "Speeding Up Automatic Hyperparameter Optimization of Deep Neural Networks by Extrapolation of Learning Curves." In: *International Joint Conference on Artificial Intelligence*. 2015, pp. 3460–3468.

[26] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. "Analysis of machine learning techniques used in behavior-based malware detection." In: *Second International Conference on Advances in Computing, Control and Telecommunication Technologies (ACT)*. IEEE. 2010, pp. 201–203.

[27] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. "Malware analysis and classification: A survey." In: *Journal of Information Security* 5.02 (2014), p. 56.

[28] Felan Carlo C Garcia, II Muga, and P Felix. "Random Forest for Malware Classification." In: *arXiv preprint arXiv:1609.07770* (2016).

[29] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. "Structural detection of android malware using embedded call graphs." In: *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. ACM. 2013, pp. 45–54.

[30] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. "Adversarial perturbations against deep neural networks for malware classification." In: *arXiv preprint arXiv:1606.04435* (2016).

[31] Kurt Hornik. "Approximation capabilities of multilayer feedforward networks." In: *Neural networks* 4.2 (1991), pp. 251–257.

[32] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." In: *Neural networks* 2.5 (1989), pp. 359–366.

[33] Wenyi Huang and Jack W Stokes. "MtNet: a multi-task neural network for dynamic malware classification." In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 399–418.

[34] Nwokedi Idika and Aditya P Mathur. "A survey of malware detection techniques." In: *Center for Education and Research in Information Assurance and Security (CERIAS)* 48 (2007).

[35] Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine Annette Shoemaker. "Hyperparameter Optimization of Deep Neural Networks Using Non-Probabilistic RBF Surrogate Model." In: *arXiv preprint arXiv:1607.08316* (2016).

[36] Rafiqul Islam, Ronghua Tian, Lynn M Batten, and Steve Versteeg. "Classification of malware based on integrated static and dynamic features." In: *Journal of Network and Computer Applications* 36.2 (2013), pp. 646–656.

[37] Rafiqul Islam, Ronghua Tian, Lynn Batten, and Steve Versteeg. "Classification of malware based on string and function feature selection." In: *Second Cybercrime and Trustworthy Computing Workshop (CTC)*. IEEE. 2010, pp. 9–17.

[38] Alekseui Grigorevich Ivakhnenko and Valentin Grigorevich Lapa. *Cybernetic predicting devices*. New York, CCM Information Corp., 1996.

[39] Alexey Grigorevich Ivakhnenko. "Polynomial theory of complex systems." In: *IEEE transactions on Systems, Man, and Cybernetics* 1.4 (1971), pp. 364–378.

[40] Qingshan Jiang, Xinxing Zhao, and Kai Huang. "A feature selection method for malware detection." In: *Information and Automation (ICIA), 2011 IEEE International Conference on*. IEEE. 2011, pp. 890–895.

[41] Ban Mohammed Khammas, Alireza Monemi, Joseph Stephen Bassi, Ismahani Ismail, Sulaiman Mohd Nor, and Muhammad Nadzir Marsono. "Feature selection and machine learning classification for malware detection." In: *Jurnal Teknologi* 77.1 (2015).

[42] Youngjoon Ki, Eunjin Kim, and Huy Kang Kim. "A novel approach to detect malware based on API call sequence analysis." In: *International Journal of Distributed Sensor Networks* 11.6 (2015).

[43] Sean Kilgallon, Leonardo De La Rosa, and John Cavazos. "Improving the Effectiveness and Efficiency of Dynamic Malware Analysis with Machine Learning." In: *International Symposium on Resilient Cyber Systems*. 2017.

[44] Sean Kilgallon, Leonardo De La Rosa, and John Cavazos. "Improving the effectiveness and efficiency of dynamic malware analysis with machine learning." In: *Resilience Week (RWS), 2017*. IEEE. 2017, pp. 30–36.

[45] Dhilung Kirat, Giovanni Vigna, and Christopher Kruegel. "BareCloud: Bare-metal Analysis-based Evasive Malware Detection." In: *USENIX Security*. Vol. 2014. 2014, pp. 287–301.

[46] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert. "Deep learning for classification of malware system call sequences." In: *Australasian Joint Conference on Artificial Intelligence*. Springer. 2016, pp. 137–149.

[47] G Bala Krishna, V Radha, and K Venugopala Rao. "Review of Contemporary Literature on Machine Learning based Malware Analysis and Detection Strategies." In: *Global Journal of Computer Science and Technology* 16.5 (2016).

[48] Oak Ridge National Laboratory. *Titan Supercomputer*. Retrieved from: https://www.olcf.ornl.gov/olcf-resources/compute-systems/titan/.

[49] Chih-Ta Lin, Nai-Jian Wang, Han Xiao, and Claudia Eckert. "Feature Selection and Extraction for Malware Classification." In: *J. Inf. Sci. Eng.* 31.3 (2015), pp. 965–992.

[50] Martina Lindorfer, Matthias Neugschwandtner, and Christian Platzer. "Marvin: Efficient and comprehensive mobile app classification through static and dynamic analysis." In: *39th Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 2. IEEE. 2015, pp. 422–433.

[51] LM Security. *Static Malware Analysis*. Retrieved from: http://resources.infosecinstitute.com/malware-analysis-basics-static-analysis/. 2017.

[52] Raymond W Lo, Karl N Levitt, and Ronald A Olsson. "MCF: A malicious code filter." In: *Computers & Security* 14.6 (1995), pp. 541–566.

[53] Ilya Loshchilov and Frank Hutter. "CMA-ES for Hyperparameter Optimization of Deep Neural Networks." In: *arXiv preprint arXiv:1604.07269* (2016).

[54] Eitan Menahem, Asaf Shabtai, Lior Rokach, and Yuval Elovici. "Improving malware detection by applying multi-inducer ensemble." In: *Computational Statistics & Data Analysis* 53.4 (2009), pp. 1483–1494.

[55] Microsoft. *TrojanDownloader:Win32/Banload*. Retrieved from: https://www.microsoft.com/en-us/wdsi/threats/malware-encyclopedia-description?Name=TrojanDownloader:Win32/Banload.

[56] Aziz Mohaisen, Omar Alrawi, and Manar Mohaisen. "Amal: High-fidelity, behavior-based automated malware analysis and classification." In: *Computers & Security* 52 (2015), pp. 251–266.

[57] Savita Mohurle and Manisha Patil. "A brief study of Wannacry Threat: Ransomware Attack 2017." In: *International Journal of Advanced Research in Computer Science* 8.5 (2017).

[58] Veelasha Moonsamy, Ronghua Tian, and Lynn Batten. "Feature reduction to speed up malware classification." In: *Nordic Conference on Secure IT Systems*. Springer. 2011, pp. 176–188.

[59] Andreas Moser, Christopher Kruegel, and Engin Kirda. "Limits of static analysis for malware detection." In: *Twenty-third Annual Computer security applications conference*. IEEE. 2007, pp. 421–430.

[60] Lakshmanan Nataraj, S Karthikeyan, Gregoire Jacob, and BS Manjunath. "Malware images: visualization and automatic classification." In: *Proceedings of the 8th international symposium on visualization for cyber security*. ACM. 2011, p. 4.

[61] Hiran V Nath and Babu M Mehtre. "Static Malware Analysis Using Machine Learning Methods." In: *International Conference on Security in Computer Networks and Distributed Systems*. Springer. 2014, pp. 440–450.

[62] Swathi Pai, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H Austin, and Mark Stamp. "Clustering for malware classification." In: *Journal of Computer Virology and Hacking Techniques* 13.2 (2017), pp. 95–107.

[63] Uğur Pehlivan, Nuray Baltaci, Cengiz Acartürk, and Nazife Baykal. "The analysis of feature selection methods and classification algorithms in permission based Android malware detection." In: *Computational Intelligence in Cyber Security (CICS), 2014 IEEE Symposium on*. IEEE. 2014, pp. 1–8.

[64] Thomas E Potok, Catherine D Schuman, Steven R Young, Robert M Patton, Federico Spedalieri, Jeremy Liu, Ke-Thia Yao, Garrett Rose, and Gangotree Chakma. "A study of complex deep learning networks on high performance, neuromorphic, and quantum computers." In: *Proceedings of the Workshop on Machine Learning in High Performance Computing Environments*. IEEE Press. 2016, pp. 47–55.

[65] Yong Qiao, Yuexiang Yang, Jie He, Chuan Tang, and Zhixue Liu. "CBM: free, automatic malware analysis framework using API call sequences." In: *Knowledge Engineering and Management*. Springer, 2014, pp. 225–236.

[66] Radare2. URL: https://www.radare.org/r/.

[67] Karthik Raman. *Towards classification of polymorphic malware*. University of California, Irvine, 2011. ISBN: 1124517820.

[68] Smita Ranveer and Swapnaja Hiray. "Comparative analysis of feature extraction methods of malware detection." In: *International Journal of Computer Applications* 120.5 (2015).

[69] ReversingLabs. URL: https://www.reversinglabs.com/.

[70] ReversingLabs. *Active File Decomposition*. Retrieved from: https://www.reversinglabs.com/technology/active-file-decomposition.html.

[71] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. "Automatic analysis of malware behavior using machine learning." In: *Journal of Computer Security* 19.4 (2011), pp. 639–668.

[72] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. "Learning representations by back-propagating errors." In: *Cognitive modeling* 5.3 (1988), p. 1.

[73] Imtithal A Saeed, Ali Selamat, and Ali MA Abuagoub. "A survey on malware and malware detection systems." In: *International Journal of Computer Applications* 67.16 (2013).

[74] Justin Sahs and Latifur Khan. "A machine learning approach to android malware detection." In: *Intelligence and security informatics conference (eisic), 2012 european*. IEEE. 2012, pp. 141–147.

[75] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. "Opcode sequences as representation of executables for data-mining-based unknown malware detection." In: *Information Sciences* 231 (2013), pp. 64–82.

[76] Igor Santos, Carlos Laorden, and Pablo G Bringas. "Collective classification for unknown malware detection." In: *Proceedings of the International Conference on Security and Cryptography (SECRYPT)*. IEEE. 2011, pp. 251–256.

[77] Joshua Saxe and Konstantin Berlin. "Deep neural network based malware detection using two dimensional binary program features." In: *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*. IEEE. 2015, pp. 11–20.

[78] Robert Searles, Lifan Xu, William Killian, Tristan Vanderbruggen, Teague Forren, John Howe, Zachary Pearson, Corey Shannon, Joshua Simmons, and John Cavazos. "Parallelization of Machine Learning Applied to Call Graphs of Binaries for Malware Detection." In: *Parallel, Distributed and Network-based Processing (PDP), 2017 25th Euromicro International Conference on*. IEEE. 2017, pp. 69–77.

[79] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. "Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey." In: *Information Security* 14.1 (2009), pp. 16–29.

[80] Priyank Singhal and Nataasha Raul. "Malware detection module using machine learning algorithms to assist in centralized security in enterprise networks." In: *ACM Computing Surveys* (2012).

[81] Michael Spreitzenbarth, Thomas Schreck, Florian Echtler, Daniel Arp, and Johannes Hoffmann. "Mobile-Sandbox: combining static and dynamic analysis with machine-learning techniques." In: *International Journal of Information Security* 14.2 (2015), pp. 141–153.

[82] Xabier Ugarte-Pedrero, Davide Balzarotti, Igor Santos, and Pablo G Bringas. "SoK: deep packer inspection: a longitudinal study of the complexity of run-time packers." In: *IEEE Symposium on Security and Privacy (SP)*. IEEE. 2015, pp. 659–673.

[83] Dolly Uppal, Vishakha Mehra, and Vinod Verma. "Basic survey on malware analysis, tools and techniques." In: *International Journal on Computational Sciences and Applications (IJCSA)* 4.1 (2014), pp. 103–120.

[84] Tristan Vanderbruggen and John Cavazos. "Large-scale exploration of feature sets and deep learning models to classify malicious applications." In: *Resilience Week (RWS), 2017*. IEEE. 2017, pp. 37–43.

[85] Mihai Vasilescu, Laura Gheorghe, and Nicolae Tapus. "Practical malware analysis based on sandboxing." In: *Networking in Education and Research Joint Conference: 13th Romanian educational and research network & 8th Research and*

Educational Networking Association of Moldova Conference. IEEE. 2014, pp. 1–6.

[86] Carsten Willems, Thorsten Holz, and Felix Freiling. "Toward automated dynamic malware analysis using cwsandbox." In: *IEEE Security & Privacy* 5.2 (2007).

[87] Suleiman Y Yerima, Sakir Sezer, and Gavin McWilliams. "Analysis of Bayesian classification-based approaches for Android malware detection." In: *IET Information Security* 8.1 (2014), pp. 25–36.

[88] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. "Android malware detection using parallel machine learning classifiers." In: *Next Generation Mobile Apps, Services and Technologies (NGMAST), 2014 Eighth International Conference on*. IEEE. 2014, pp. 37–42.

[89] Suleiman Y Yerima, Sakir Sezer, and Igor Muttik. "High accuracy android malware detection using ensemble learning." In: *IET Information Security* 9.6 (2015), pp. 313–320.

[90] Xinchuan Zeng and Tony R Martinez. "Distribution-balanced stratified cross-validation for accuracy estimation." In: *Journal of Experimental & Theoretical Artificial Intelligence* 12.1 (2000), pp. 1–12.