5-21-2020

Machine Learning Interpretability in Malware Detection

William Ryan Brigugilio
University of Windsor

Machine Learning Interpretability in Malware Detection

By

William R. Briguglio

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2020

Machine Learning Interpretability in Malware Detection

by

William R. Briguglio

APPROVED BY:

M. Mirhassani
Department of Electrical and Computer Engineering

A. Ngom
School of Computer Science

S. Sherif, Advisor
School of Computer Science

March 17, 2019

DECLARATION OF CO-AUTHORSHIP AND PREVIOUS PUBLICATION

I. Co-Authorship

I hereby declare that this thesis incorporates material that is the result of joint research, as follows: Chapter 2 of the thesis was co-authored with Dr. Saad and Dr. Elmiligi. Chapter 3 was co-authored with Dr. Saad and M.Sc. Farhan Mahmood and Dr. Elmiligi. Chapter 4 was co-authored with Dr. Saad. In the case of chapter 2, which was a position paper, the position was principally arrived at by Dr. Saad and Dr. Elmiligi while I read and summarized relevant literature and helped formulate the final wording in which the position was put forward. For Chapter 3, Dr. Saad contributed the objectives of the paper and important guidance for achieving said objectives as well as proof reading and review of the paper. The implementation of the malicious functionalities of the malware described therein, and the web application used to test said malware, was completed in equal parts by Farhan Mahmood and myself. Farhan also selected various malware detection tools as well as designed and conducted the tests to evaluate the the malware's ability to bypass said detectors. In the case of chapter 4 the key ideas, primary contributions, experimental designs, data analysis, interpretation, and writing were performed by myself, and the contribution of the co-author, Dr. Saad, was primarily through valuable guidance and feedback on refinement of ideas and editing of the paper.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work.

II. Previous Publication

This thesis includes 3 original papers that have been previously published/submitted for publication in peer reviewed journals, as follows:

Thesis Chapter	Publication title/full citation	Publication Status
Chapter 2	Saad, S.; Briguglio, W. and Elmiligi, H. (2019). The Curious Case of Machine Learning in Malware Detection. In Proceedings of the 5th International Conference on Information Systems Security and Privacy - Volume 1: ICISSP, ISBN 978-989-758-359-9, pages 528-535. DOI: 10.5220/0007470705280535	Published
Chapter 3	Saad, S.; Mahmood, F.; Briguglio, W. and Elmiligi, H. (2019) JSLess: A Tale of a Fileless Javascript Memory-Resident Malware. In Proceedings of the 15th International Conference on Information Security Practice and Experience - Volume 1: ISBN 978-3-030-34338-5, pages 113-131. DOI: 10.1007/978-3-030-34339-2	Published
Chapter 4	Briguglio, W. and Saad, S. (2019). Interpreting Machine Learning Malware Detectors Which Leverage N-gram Analysis. In Proceedings of the 12th International Symposium on Foundations and Practice of Security	In Press

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as a graduate student at the University of Windsor

III. General

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard

referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

The ever increasing processing power of modern computers, as well as the increased availability of large and complex data sets, has led to an explosion in machine learning research. This has led to increasingly complex machine learning algorithms, such as Convolutional Neural Networks, with increasingly complex applications, such as malware detection.

Recently, malware authors have become increasingly successful in bypassing traditional malware detection methods, partly due to advanced evasion techniques such as obfuscation and server-side polymorphism. Further, new programming paradigms such as fileless malware, that is malware that exist only in the main memory (RAM) of the infected host, add to the challenges faced with modern day malware detection. This has led security specialists to turn to machine learning to augment their malware detection systems. However, with this new technology comes new challenges. One of these challenges is the need for interpretability in machine learning.

Machine learning interpretability is the process of giving explanations of a machine learning model's predictions to humans. Rather than trying to understand everything that is learnt by the model, it is an attempt to find intuitive explanations which are simple enough and provide relevant information for downstream tasks. Cybersecurity analysts always prefer interpretable solutions because of the need to fine tune these solutions. If malware analysts can't interpret the reason behind a misclassification, they will not accept the non-interpretable or "black box" detector.

In this thesis, we provide an overview of machine learning and discuss its roll in cyber security, the challenges it faces, and potential improvements to current approaches in the literature. We showcase its necessity as a result of new computing paradigms by implementing a proof of concept fileless malware with JavaScript. We then present techniques for interpreting machine learning based detectors which leverage n-gram analysis and put forward a novel and fully interpretable approach for malware detection which uses convolutional neural networks. We also define a novel approach for evaluating the robustness of a machine learning based detector.

ACKNOWLEDGEMENTS

Here I would like to acknowledge the invaluable mentorship of my supervisor Dr. Sherif, not only during my masters but also during the last year of my undergrad, during which I would not have considered graduate school were it not for his encouragement.

TABLE OF CONTENTS

DECLARATION OF CO-AUTHORSHIP AND PREVIOUS PUBLICATION iii

ABSTRACT vi

ACKNOWLEDGEMENTS vii

LIST OF TABLES xi

LIST OF FIGURES xii

1 Introduction 1
- 1.1 Machine Learning Algorithms . . . 3
- 1.2 Machine Learning Interpretability . . . 7
- 1.3 Machine Learning in Cyber Security . . . 9
- 1.4 Thesis Overview . . . 9
- References . . . 10

2 The Curious Case of Machine Learning in Malware Detection 11
- 2.1 Introduction . . . 11
- 2.2 Literature Review . . . 13
- 2.3 Emerging Malware Threats . . . 15
  - 2.3.1 Unconventional Computing Paradigms . . . 15
  - 2.3.2 Unconventional Evasion Techniques . . . 17
- 2.4 Practical Challenges . . . 20
  - 2.4.1 Cost of Training Detectors . . . 20
  - 2.4.2 Malware Detector Interpretability . . . 21
  - 2.4.3 Adversarial Malware . . . 21
- 2.5 Bridging the Detection Gap . . . 22
  - 2.5.1 Disposable Micro Detectors . . . 22
  - 2.5.2 Analyst Friendly Interpretation . . . 23
  - 2.5.3 Anti Adversarial Malware . . . 24
- 2.6 Conclusion . . . 25
- References . . . 25

3 JSLess: A Tale of Fileless JavaScript Memory-Resident Malware 31
- 3.1 Introduction . . . 31
- 3.2 Literature Review . . . 34
- 3.3 Benign Features with Malicious Potentials . . . 37
  - 3.3.1 WebSockets . . . 37
  - 3.3.2 WebWorker . . . 38
  - 3.3.3 ServiceWorker . . . 39
- 3.4 JavaScript Fileless Malware . . . 40

3.4.1 Infection Scenarios . . . 40
3.4.2 Operational Scenarios . . . 42
3.4.3 Attack Vectors . . . 46
3.4.3.1 Data Stealing . . . 46
3.4.3.2 DDoS . . . 47
3.4.3.3 Resource Consumption Attack . . . 47
3.5 Experiment & Evaluation . . . 47
3.5.1 JS Malware Detection Tools . . . 48
3.5.2 Detection & Mitigation . . . 48
3.5.3 Detection Tool Analysis Results . . . 49
3.6 Conclusion & Future Work . . . 51
References . . . 52

4 Interpreting Machine Learning Malware Detectors Which Leverage N-gram Analysis 56
4.1 Introduction . . . 56
4.2 Literature Review . . . 58
4.3 Method . . . 61
4.4 Interpretation . . . 64
4.4.1 Logistic Regression Model Interpretation . . . 64
4.4.2 Random Forest Interpretation . . . 69
4.4.3 Neural Network Model Interpretation . . . 72
4.5 Conclusion . . . 77
References . . . 78

5 Interpreting Machine Learning Malware Detectors Which Leverage Convolutional Neural 81
5.1 Introduction . . . 81
5.2 Literature Review . . . 83
5.3 Method . . . 87
5.4 Results . . . 91
5.4.1 Evaluation of Our Model . . . 91
5.4.2 Interpretation . . . 91
5.4.3 Comparison With Other Models . . . 96
5.5 Conclusion . . . 98
References . . . 99

6 Robustness Metric 102
6.1 Introduction . . . 102
6.2 The Robustness Metric . . . 104
6.3 Method . . . 106
6.4 Conclusion . . . 109
References . . . 109

7 Conclusion 111

VITA AUCTORIS **113**

LIST OF TABLES

1.1.1 Machine Learning Further Readering 7

3.5.1 JavaScript and Web App Malware Detection Tools 48

4.3.1 Class distribution in Data Set 61

4.4.1 Max 15 Absolute Weights of the Logistic Regression Model Averaged Across All 9 Binary Sub-classifiers 65

4.4.2 Max 15 Weights for Kelihos_ver3 Binary Sub-classifier 66

4.4.3 Max Feature 15 Importances for Random Forest 70

4.4.4 Max Average Absolute Relevances 73

4.4.5 Max Avg Relevances for Class 3 74

4.4.6 Layer 1 Nodes relevance to n_{40}^2 76

5.3.1 Class distribution in Data Set 88

5.4.1 Confusion Matrix for Our Model on the Left Out Test Set 96

5.4.2 Confusion Matrix for Model used in [3]* 96

5.4.3 Confusion Matrix for Model used in [6]* 97

5.4.4 Model Comparisons 98

6.3.1 Robustness Metrics For Trained Models 107

LIST OF FIGURES

1.1.1 Neural Network with 6 nodes in the input layer, 6 in the hidden layer, and 3 in the output layer 7

3.4.1 JavaScript Fileless Malware First Infection Scenario 41
3.4.2 JavaScript Fileless Malware Second Infection Scenario 42
3.4.3 Obfuscated JavaScript code injection 44

5.3.1 Images from two samples from each of the 6 classes. Images in columns labeled F are the first three channels of the samples interpreted as RGBchannels while the images in columns labeled L are the last three channels 90
5.3.2 Model Architecture 90
5.4.1 Test and Validation Loss History 92
5.4.2 Test and Validation Balanced Categorical Accuracy History 92
5.4.3 0AnoOZDNbPXIr2MRBSCJ Relevance Map 93
5.4.4 Terminal output when working backwards from the relevant 6-grams to the code snippets 94

6.3.1 Robustness plots with balanced accuracy on the y-axis and number of features deactivated on the x-axis 108

CHAPTER 1

Introduction

Machine Learning has come a long way in recent years. There has been a vast amount of papers published in the last decade which offer a number of substantial improvements on machine learning algorithms both old and new. Along side this research there has also been many papers which study the various applications of machine learning algorithms. From perhaps the most well known, even among non-experts, such as machine vision and natural language processing, to the less well known but all the while pervasive and significant medical, commercial, and industrial applications.

Machine learning is the study of algorithms which allow computers to preform a specific task without the use of explicit programming. These algorithms accomplish this by using statistics and inference techniques to detect or "learn" patterns within data (hence machine learning is considered a type of pattern recognition). These patterns are then used in various ways to preform a certain task.

The data used by the machine learning algorithm, referred to as the "data set", is typically a table where each row is considered a single record or "sample" and each column corresponds to a "feature" whose values describe said feature for each sample in the data set. For example, a housing market data set would have a separate row for each house and a column for each feature, such as the number of rooms, square footage, and so on. In practice, a data set of n samples and m features is an $n \times m$ array (table, matrix, etc.) of values where the i^{th} row of the feature array gives the "feature vector" of the i^{th} sample. Meanwhile, the j^{th} column in the i^{th} row gives the

value of the j^{th} feature for the i^{th} sample. This value is denoted denoted $x_{i,j}$ however the row index i is omitted when the context makes it obvious we are discussing a single example. The processes of choosing features and determining their value for each sample are known as feature selection and feature extraction respectively. In some cases, such as computer vision with neural networks, feature extraction is done by the machine learning algorithm automatically from raw data such as images (e.g. pixel RGB values).

Machine learning can be broadly separated into two categories, supervised and unsupervised learning. In supervised learning, each sample is accompanied by a "label", and the $1 \times n$ array of labels forms a column vector where the i^{th} element, denoted y_i, is the label of the sample i^{th} sample. The machine learning algorithm uses a subset of the data set, called the training set, in order to learn the relationship between the feature values and the label. This learnt relationship is referred to as "the model" and can be used to predict the label of previously unseen samples from their feature values. However, before being used as a predictor, to ensure the model is effective at making predictions, the trained model makes predictions on a portion of the labeled data set which was left out during training, called the test set. The predicted labels are compared with the known labels to see if the predictions are reliable. What is considered reliable is application specific, and as we shall see later, goes far beyond simple metrics such as accuracy.

Supervised learning can be further divided into two subcategories, regression and classification. In the former case, the model predicts a real value such as a the price a house will sell for. In the latter case, the model predicts a discrete value which corresponds to the class of a sample. For example, in binary classification the model may predict either 0 or 1 which may correspond to the rejection class or approval class of mortgage applicants based on features extracted from financial histories.

In the case of unsupervised learning, samples in the data set are not accompanied by labels and here the objective is to find unknown structure and relationships in the data. Models trained using unsupervised learning can find anomalous data points or uncover potential classes to be used later in a supervised approach. We hold off on a

discussion of unsupervised machine learning techniques as they are not the focus of this thesis.

Next we introduce a high level discussion of the machine learning models used later in this thesis, specifically in Chapter 4. This will serve as a preliminary for understanding what is discussed there for readers without a background in machine learning.

1.1 Machine Learning Algorithms

One of the most basic and widely used machine learning algorithms is Linear Regression. Linear Regression models are used for regression by simply calculating a weighted sum of a sample's feature values and adding a real valued "bias" which yields it's predicted label. The weight of the j^{th} feature is denoted w_j resulting in the following equation for predicting a samples label:

$$y = \sum_{j=1}^{m} x_j w_j + \beta \tag{1}$$

A version of Linear Regression adapted for binary classification is Logistic Regression in which the result of the sigmoid function applied to the weighted sum is used to determine the label. Since the sigmoid function outputs only values between 0 and 1, a sample is predicted to belong to class 0 if it produces an output less than 0.5, otherwise it is predicted to belong to class 1. In the case where there are more then two classes, a Logistic Regression model is trained for each class. For the k^{th} Logistic Regression model, only the samples belong to the k^{th} class are labeled 1 and the rest are labeled 0. Thus the multi-class scenario is treated as multiple binary classification scenarios. During prediction, a sample is predicted as the class whose Logistic Regression model produced the highest output. This is known as one-vs-rest classification. Below is the function a logistic regression model uses to make predictions.

$$y = \frac{1}{1 + e^{-z}} \quad \text{where } z \text{ is the result of equation 1} \tag{2}$$

So far we have seen how linear regression and logistic regression models make predictions, but as of yet we have not discussed how they are trained. Before we can do this we must introduce the idea of the "loss function". The loss function provides a measure of the error in a models predictions given a set of labeled samples and must be differentiable with respect to the parameters of the model which we wish to learn. Here parameters refers to any values which we learn during training, such as feature weights and the bias. (Values which are not learnt during training and are chosen before hand are called hyper-parameters.) When we say "differentiable with respect to the parameters we wish to learn", we are saying that we can determine if increasing one of the parameters by a very small amount will cause the loss function to decrease or increase. For a more detailed discussion of linear and logistic regression, we refer the reader to [2]

Since the loss function is a measure of our model's error, and we wish to minimize the error, we also wish to minimize the loss function. Therefore, we determine for each trainable parameter whether increasing it or decreasing it will cause the loss function to decrease and we add/subtract a very small amount to/from the parameter's value based off this. Once we do this for all parameters, we recalculate the loss function and repeat. Since we cannot jump directly to the values where the loss function is minimized, because the derivative only gives local information about the parameters effect on the output of the loss function, we must take small steps each iteration and repeat. In this way, making locally optimal decisions with course corrections along the way, we ideally arrive at a "global minimum". That is, the values for the parameters of the model for which the loss function is lowest given the training set. The mathematical construct which specifies the direction to move each parameter in order to minimize the loss function is called the "gradient" and since we use this gradient to descend the loss function, we call this process gradient descent. There are many things to consider when preforming gradient descent, such as the size of changes to parameters at each iteration, but we refer the reader to [3] for a more thorough and detailed discussion.

Another classic machine learning algorithm is the decision tree. Decision trees

can be used for regression or classification but here we will focus on the classification case. Decision trees are made up of three parts; a root node, where the decision process starts, leaf nodes, which are at the other end of the tree opposite of the root node, and internal nodes, which lay along the path from root to leaf node. There is only one path from the root node to any given leaf node. During prediction, the algorithm moves from the root node, through some internal nodes to a leaf, and the value of said leaf node determines the predicted class of the sample. The path which the algorithm takes is determined by the value of the sample's features and "split conditions" in each of the root and internal nodes, such as $x_j > 13$.

Decision trees are trained by using a labeled training set to determine the split condition at each node which maximizes the "information gain" in the child nodes. Starting at the root node which is reachable by all training samples, a split is chosen which best separates the classes within the training set. This is repeated at the child nodes of the root node, and repeated again for their children and so on, until the child nodes produced by a splitting condition are only reachable by training examples which all belong to the same class, or some other stopping condition is met, such as maximum depth of the tree. In the case where the leaf node is reachable by training samples from more then one class, the prediction is the class which the majority of those training samples belong to.

An extension of the decision tree algorithm is the Random Forest algorithm which trains a large number of decision trees on random subsets of the training set, using random subsets of the feature set. At prediction time, each constituent decision tree predicts the class of the sample which counts as a vote for said class. The votes from all the sub trees are counted and the class with the most votes is the prediction of the Random Forest. This is known as an ensemble method, and its strength is that misclassification occurs only if the majority of the constituent decision trees make a misclassification. Further, since the constituent decision trees all use different feature sets and were trained with different training sets, a feature value which is uncommon for one class will not trick all of the constituent decision trees. The end result is a model which is more stable when encountering unseen data, achieving

better performance. This is known as generalizability. We say a model generalizes well when it achieves performance which is equal or better with unseen data as with the training data. For a deeper discussion on random forests we again refer the reader to [2]

The last machine learning algorithm which we'll introduce here is the Neural Network. Neural networks are much more complex then the algorithms discussed so far but some of the ideas are the same. The standard neural network architecture is the feed forward architecture such as the one shown in figure 1.1.1. This type of neural network makes predictions as follows. The feature values are inputted into their respective input neuron (or node) in the first layer, called the input layer. (i.e. feature x_j is inputted to the j^{th} neuron in the input layer.) Next, this value is propagated along the connections (shown in 1.1.1 as lines connecting the neuron (circles) in different layers) from the input layer to the first hidden layer. When the neurons in the next layer receive the values from the previous layer along these connections, they multiply them by the weight associated with each connection and sum the results along with a bias. This weighted sum is identical to that calculated in equation 1 for the linear regression algorithm, except that the feature values are replaced by the outputs of neurons in the previous layer. Each neuron then applies an "activation function", of which there are many different types, such as the sigmoid function from equation 2. The output of the activation applied to the weighted sum of incoming signals from the previous layer is the output of that node, which is sent to the next layer. This process is repeated for an arbitrary number of layers until the final layer outputs a value which indicates the predicted class.

The way neural networks are trained is also similar to the way the linear and logistic regression models were trained. Except here there loss function is a complex composite function where the derivative must be taken with respect to many more parameters, in some cases well into the tens of millions. However, the basic idea of gradient descent still applies, incrementally make very small changes to the weights in a direction which decreases the loss function given a training set, and repeat until we find some minimum. There are many intricacies to workout when implementing

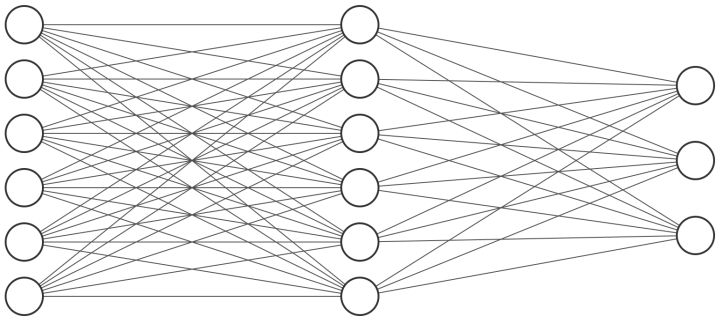

Fig. 1.1.1: Neural Network with 6 nodes in the input layer, 6 in the hidden layer, and 3 in the output layer

such a method but we do not busy our selves with them here. For a more detailed discussion of neural networks, architecture and activation function choices, and loss functions, we refer the reader to [1].

This concludes our preliminary discussion of machine learning. Table 1.1.1 summarizes more in depth sources on the topics briefly discussed here, which the reader can make use of at their own discretion.

Table 1.1.1: Machine Learning Further Readering

Logistic Regression	[2]
Linear Regression	[2]
Gradient Descent	[3]
Random Forest	[2]
Neural Networks	[1]

1.2 Machine Learning Interpretability

In the previous section we discuss the basics of machine learning and introduced some of more well known machine learning algorithms. We discuss how the machine learning algorithms are trained and how they make predictions but we now introduce one

of the central topics of this thesis, interpretability. Machine Learning interpretability is the process of giving explanations of a models predictions to humans. It is not an attempt to understand every thing that is learnt by the model, rather it an attempt to find intuitive explanations which are simple enough and give information pertinent to down stream tasks. For example, sometimes we wish to select the best feature set for a given model, in this case we would like to determine the most significant features for feature selection. The processes which determine the feature significances, of which there are many, is a simple form of machine learning interpretation.

The perfect interpretation of a machine learning model tells us only what we need to know while leaving out details of the model itself. It leaves out the messy complexities of the model in order to give us enough information for later tasks. It is usually not possible to get clear, high fidelity interpretations. Most times we have to settle for more vague interpretations such as "this input is more influential then the others" or "the model will still be accurate with such and such features no longer behaving in an informative way".

In terms of varieties of interpretations, they can generally be divided into two groups. Local interpretations are interpretations which apply only to a single sample or a subset of the sample space. Meanwhile Global interpretations apply to the entire sample space. Interpretation techniques can also be divided into categories. Model agnostic, that is techniques which can be applied to any type of machine learning model, and model specific which can only be applied to a single type of machine learning model. To be clear, global vs. local is a categorization of the interpretations themselves, while model-agnostic vs. model-specific is a categorization of the techniques used to arrive at interpretations.

One last note, since interpretation is done for the sake of down stream tasks, and further, it is influenced by the type of model and features used, this makes machine learning interpretation a application specific problem. Hence, the techniques on exposition in this thesis may be applicable with varying amounts of modification in other domains, but they are chiefly applicable in the malware detection domain.

1.3 Machine Learning in Cyber Security

Machine Learning's application in the cyber security domain is discussed at length in the following chapter so we give only a brief overview as a primer for the discussion to come, as well as to elucidate the connection between the various chapters in this thesis.

Recently, there has been increased ability for malware authors to bypass traditional malware detection methods. Techniques such as obfuscation and server-side polymorphism can help authors automatically change malware to be unrecognizable enough to bypass simple detection techniques. Further, new programming paradigms such as fileless malware, that is malware that does not exist on the file system of the infected machine, mean that new malware may not leave behind binaries to study and be used with traditional detection methods. This has lead security specialists to turn to machine learning algorithms to augment malware detection systems. However, with this new detection technique comes new challenges, one of which and the focus of this thesis, is the need for interpretable machine learning methods.

1.4 Thesis Overview

The remainder of this thesis is laid out as follows. In Chapter 2, we discuss Machine Learning's roll in cyber security, the challenges presented with its application in malware detection, and some potential improvements to the current approaches in the literature. Further we discuss the necessity for machine learning based malware detection as a result of new computing paradigms such as fileless malware, among other emerging threats. In Chapter 3, we provide a proof of concept fileless malware which uses benign functionality of JavaScript in order to carry out its malicious actions. We then test various malware detection softwares against out malware in order to show the severity of the threat of fileless malware. In Chapter 4 we present techniques for interpreting machine learning based malware detection models which leverage n-gram analysis. In Chapter 5 we put forward a novel and fully interpretable approach for

malware detection which leverages convolutional neural networks. In Chapter 6 we define a novel approach for evaluating the robustness of a machine learning based malware detector based off the features it uses. Lastly, we end off in Chapter 7 with our conclusions and a discussion of future work.

References

[1] C. C. Aggarwal. *Neural Networks and Deep Learning*. Springer, 2018. ISBN: 978-3-319-94462-3. DOI: 10.1007/978-3-319-94463-0.

[2] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. Springer, 2009. URL: http://www-stat.stanford.edu/~tibs/ElemStatLearn/.

[3] S. Ruder. "An overview of gradient descent optimization algorithms". In: *CoRR* abs/1609.04747 (2016). arXiv: 1609.04747. URL: http://arxiv.org/abs/1609.04747.

CHAPTER 2

The Curious Case of Machine Learning in Malware Detection

Sherif Saad, William Briguglio, and Haytham Elmiligi

In Proceedings of the 5th International Conference on Information Systems Security and Privacy

2.1 Introduction

Nowadays, computer networks and the Internet have become the primary tool for spreading and distributing malware by malware authors. The massive number of feature-rich programming languages and off-the-shelf software libraries enable the development of new sophisticated malware such as botnet, fileless, k-ary and ransomware. New computing paradigms, such as cloud computing and the Internet of Things, expand potential malware infection sites from PC's to any electronic device.

To decide if software code is malicious or benign, we could either use static analysis or dynamic analysis. Static analysis techniques do not execute the code and only examine the code structure and other binary data properties. Dynamic analysis techniques, on the other hand, execute the code to observe the execution behaviors of the code over the network or at end-point devices. Some malware detection systems apply only static or dynamic techniques, and some apply both. While dynamic malware analysis techniques are not intended to replace static analysis techniques, recent unconventional malware attacks (botnet, ransomware, fileless, etc.) and the use of sophisticated evasion techniques to avoid detection have shown the urgent need of dynamic analysis and the limitations of static analysis. In our opinion, the use of

dynamic and behavioral malware analysis will dominate the next-generation malware detection systems.

There is a general belief among cybersecurity experts that antimalware tools and systems powered by artificial intelligence and machine learning will be the solution to modern malware attacks. The number of studies published in the last few years on malware detection techniques that leverage machine learning is a distinct evidence of this belief as shown in section 2.2. In the literature, various malware detection techniques using machine learning are proposed with excellent detection accuracy. However, malware attacks in the wild continue to grow and manage to bypass malware detection systems powered by machine learning techniques. This is because it is difficult to operate and deploy machine learning for malware detection in a production environment or the performance in a production environment is disturbing (e.g. high false positives rate). In fact, there is a significant difference (a detection gap) between the accuracy of malware detection techniques in the literature and their accuracy in a production environment.

A perfect malware detection system will detect all types of malicious software and will never consider a benign software as a malicious one. Cohen provided a formal proof that creating a perfect malware detection system is not possible [7, 6]. Moreover, Chess and White proved that a malware detector with zero false positives is not possible [4]. Selcuk et al. discussed the undecidable problems in malware detection in more details [31]. In light of this, the high levels of accuracy claimed by commercial malware detection systems and some malware detection studies in literature seems questionable.

In this Chapter, we briefly review the current state of the art in malware detection using machine learning approaches. Then, we discuss the importance of dynamic and behavioral analysis based on emerging malware threats. Next, the shortcomings of the current machine learning malware detectors are explained to indicate their limitations in the wild. Finally, we discuss the possible solutions to improve the quality of malware detection systems and point out potential research directions.

2.2 Literature Review

In recent years, machine learning algorithms have been used to design both static and dynamic analysis techniques for malware detection. Hassen et al. proposed a new technique for malware classification using static analysis based on control statement shingling [15]. In their work, they used static analysis to classify malware instances into new or known malware families. They extracted features from disassembled malicious binaries and used the random forest algorithm to classify malware using the extracted features. Using a dataset of 10,260 malware instances, they reported up to 99.21% accuracy.

Static analysis has been used to study malwares that infect embedded systems, mobile devices, and other IoT devices. Naeem et al. proposed a static analysis technique to detect IoT malware [24]. The proposed technique converts a malware file to a grayscale image and extracts a set of visual features from the malware image to train an SVM classifier that could distinguish between malware families using visual features. Using a dataset of 9342 samples that belong to 25 malware families, they reported 97.4% accuracy. Su et al. proposed a similar technique to classify IoT malware into malware families using visual features and image recognition [34]. Their approach is very similar to the one proposed in [24]. They used a one-class SVM classifier and tested their approach on IoT malwarethat infect Linux-like IoT systems; they reported 94.0% accuracy for detecting malware and 81.8% accuracy for detecting malware families. Raff et al. proposed a malware detection technique using static analysis and deep learning [29]. The proposed technique achieved 94.0% detection accuracy.

Several works have been proposed to detect Android malware apps using static analysis techniques. Sahin et al. proposed an Android malware detection model that uses app permission to detect malicious apps [26]. They used the permissions required by the app with a weighted distance function and kNN plus Naive Bayes classifier to detect malicious apps. They reported an accuracy up to 93.27%. Su and Fung used sensitive functions and app permissions to detect Android malware [35]. They used

different machine learning algorithms such as SVM, decision tree, and kNN to build an android malware detector. They reported an average accuracy between 85.0% and 90.0%

Collecting and monitoring all malware behaviors is a complicated and time consuming process. For that reason, several works in the literature focused on collecting partial dynamic behaviors of the malware. Lim et al. [19] proposed a malware detection technique by analyzing network traffic generated when the malware communicates with a malicious C&C server such as in the case of botnet or ransomware. The proposed technique extracts a set of features from network flows to present a flows sequence. The authors used different sequence alignment algorithms to classify malware traffic. They reported an accuracy above 60% when analyzing malware traffic in a real network environment.

Kilgallon et al. applied machine learning and dynamic malware analysis [17]. The proposed technique gathers register value information and API calls made by the monitored malware binaries. The collected information is stored in vector structures and analyzed using a value set analysis method. Then, they used a linear similarity metric to compare unseen malware to known malware binaries. Their experiment showed that the proposed technique could detect malware with an accuracy up to 98.0%

Omind and Nathan proposed a behavioral-based malware detection method using a deep belief network [9]. The proposed method collected data about malware behaviors from a sandbox environment. The collected data is API calls, registry entries, visited websites, accessed ports, and IP addresses. Then using a deep neural network of eight layers, it generates malware signatures. These signatures could be used to train malware detectors. In their experiments, they reported up to 95.3% detection accuracy with a malware detector utilizing the SVM algorithm.

Yeo et al. proposed a new malware detection method by monitoring malicious behaviors in network traffic [38]. They designed 35 features to describe malicious traffic of malware instances. They tested several machine learning algorithms including CNN, MLP, SVM, and random forest. The proposed method achieved an accuracy

above 85% when utilizing CNN or random forest. Prokofiev et al. proposed a machine learning technique to detect C&C traffic of infected IoT devices [28]. The proposed approach used network traffic features such as port number, IP addresses, connection duration and frequency. They reported a detection accuracy up to 97.3%. However, the proposed approach is still relying on traditional malware analysis methods and will not be able to work in production IoT deployment as discussed in [33]. Several hybrid malware detection techniques that combine both static and dynamic analysis have also been proposed [21, 27]. These techniques try to improve the quality and performance of malware detection systems by taking advantage of static and dynamic analysis to build robust malware detection systems.

2.3 Emerging Malware Threats

With the recent changes in malware development and the rise of commercial malware (malicious code rented or purchased), many new challenges are facing malware analysts that make static analysis more difficult and impractical. These challenges will force anti malware vendors to adapt behavioral malware analysis and detection techniques. In our opinion, there are two main reasons behind these challenges; the rise of unconventional computing paradigms and unconventional evasion techniques. There is a new generation of malware that take advantage of unconventional computing paradigms and off-the-shelf soft-ware libraries written by feature-rich programming languages. The current state-of-the-art malware analysis/detection techniques and tools are not effective against this new generation of malware.

2.3.1 Unconventional Computing Paradigms

New computing paradigms and technologies such as cloud computing, the internet of things, big data, in-memory computing, and blockchain introduced new playgrounds for malware authors to develop com-plex and sophisticated malwares that are almost un-detectable. Here we describe several recent examples of new malware threats that are difficult to detect or analyze using static analysis.

For instance, the Internet of Things (IoT) is an appealing platform for modern and sophisticated malware such as ransomware. Zhang-Kennedy et al. discussed the ransomware threat in IoT and how a self-spreading ransomware could infect an IoT ecosystem [39]. The authors pointed out that the ransomware will mainly lock down IoT devices and disable the essential functions of these devices. The study focused on identifying the attack vectors in IoT, the techniques for ransomware self-spreading in IoT, and predicting the most likely class of IoT applications to be a target for ransomware attacks. Finally, the authors identified the techniques the ransomware could apply to lock down IoT devices. Authors in [39] used a Raspberry Pi to develop a proof of concept IoT ransomware that can infect an IoT system. One interesting aspect in [39] is the need for collaboration or swarming behavior in IoT ransomware, where the IoT ransomware will spread as much as possible and then lock down the devices or device and then spread.

Miller and Valasek developed a proof-of-concept for malicious code that infects connected cars and lockdowns key functions [22].For instance, the authors demonstrated the ability for the malicious code to control the steering wheel of a vehicle, disable the breaks, lock doors, and shut down the engine while in motion. Behaving as ransomware, this real example of a malware that locks and disables key features in IoT systems (e.g. connected cars) could have life threatening consequences if the ransom is not paid. The study explained a design flow in the Controller Area Network (CAN) protocol that allows malicious and crafted CAN messages to be injected into the vehicle CAN channel by a compromised mobile phone that is connected to the vehicle entertainment unit. It was reported that for some vehicles only the dealership could restore and patch the vehicle to prevent this attack. Choi et al. proposed a solution for malware attacks in connected vehicles using machine learning [5]. The solution uses SVM to distinguish between crafted malicious CAN messages, and benign CAN messages generated by actual electronic control units (ECU). The model extracts features from the vehicle ECUs and creates fingerprints for those ECUs. The ECU fingerprint is noticeable in a benign CAN message and does not exist in a malicious message.

Azmoodeh et al. discussed a new technique to detect ransomware attacks in IoT systems by monitoring the energy consumption of infected devices [3]. As a proof of concept, they studied the energy consumption of infected Android devices. The devices were infected by a ransomware with crypto impact. They used different machine learning models (kNN, SVM, NN, and Random Forest) to analyze energy consumption data and extract unique patterns to detect compromised Android devices. They reported a ransomware detection accuracy of 95.65%.

In 2015, Karam (INTERPOL) and Kamluk (Kaspersky lab) introduced a proof of concept distributed malware that also takes advantage of blockchain technology [16]. In 2018, Moubarak et al. provided design and implementation of a K-ary malware (distributed malware) that takes advantages of the blockchain networks such as Etherum and Hyperledger [23]. The proposed malware is stored and executed inside blockchain networks and acts as a malicious keylogger. While detecting a K-ary malware is an NP-hard problem [10], it is also complicated to implement a K-ary malware. However, Mubarak's works demonstrated the simplicity of K-ary malware development by taking advantage of blockchain technology as a distributed and decentralized network.

2.3.2 Unconventional Evasion Techniques

The new generation of malware will use advanced evasion techniques to avoid detection by antimalware systems and tools. New evasion techniques implemented by malware authors use new technologies and off-the-shelf software libraries that enable the design of sophisticated evasion methods. Antimalware vendors and malware researchers discussed recent examples of using new antimalware evasion techniques in the wild.

Fileless malware or memory-resident malware is the new technique used by malware authors to develop and execute malicious attacks. Fileless malware resides in device memory and does not leave any files on the infected device file system. This makes the detection of the fileless malware using signature-based detection or static analysis infeasible. In addition, the fileless malware takes advantage of the utilities

and libraries that already exist in the platform of the infected device to complete its malicious intents. In other words, benign applications and software libraries are manipulated by fileless malware to accomplish the attack objectives.

Fileless malware attacks and incidents are already observed in the wild compromising large enterprises. According to KASPERSKY lab, 140 enterprises were attacked in 2017 using fileless malwares [11]. Ponemon Institute reported that 77% of the attacks against companies use fileless techniques [36]. Moreover, there are several signs that ransomware attacks are going fileless, as discussed in [20]. Besides these signs, there are other reasons in our opinion that confirms that ransomware and other malware attacks will be fileless. One main reason is the moving towards in-memory computing.

In recent years, in-memory computing and in-memory data stores became the first backbone and storage technology for many organizations. Many bigdata platforms and data grids (Apache Spark, Redis, HazelCast, etc.) enable storing data in memory for performance and scalability requirements. Valuable data and information is stored in memory for a longtime before moving to a persistent data store. In-Memory ransomware that encrypts in-memory data (such as recent transactions, financial information, etc.) present a severe and aggressive attack. This is because any attempt to reset or reboot the machine to remove the ransomware from the device memory or shutdown the application will result in losing this valuable data permanently.

The moving towards distributed and decentralized computing is another reason for the rise of fileless ransomware. In distributed and decentralized computing several nodes and devices are available to store the in-memory malware, which will increase the life expectancy of the malware since there will always be a group of active nodes were the malware could replicate and store itself.

The recent and massive development in machine learning/artificial intelligence (aka data science) and a large number of off-the-shelf machine learning libraries enable malware authors to develop advanced evasion techniques. Rigaki and Garcia proposed the use of deep learning techniques to create malicious malware samples that evade detection by mimicking the behaviors of benign applications [30]. In their work, a

proof of concept was proposed to demonstrate how malware authors could cover the malware C&C traffic. The authors use a Generative Adversarial Networks (GANs) to enable malware (e.g., botnet) to mimic the traffic of a legitimate application and avoid detection. The study showed that it is possible to modify the source code of malware to receive parameters from a GAN to change the behaviors of its C&C traffic to mimic the behaviors of other legitimate network applications, such as Facebook traffic. The enhanced malware samples were tested against the Stratosphere Linux IPS (slips) system, which uses machine learning to detect malicious traffic. The experiment showed that 63.42% of the malicious traffic was able to bypass the detection.

A research team from IBM demonstrated the use of artificial intelligence to engineering malware attacks [8]. In their study, the authors proposed DeepLocker as a proof of concept to show how next-generation malware could leverage artificial intelligence. DeepLocker is a malware generation engine that malware authors could use to empower traditional malware samples such as WannaCry with artificial intelligence. A deep convolutional neural network (CNN) was used to customize a malware attack by combining a benign application and a malware sample to generate a hybrid malware that bypasses detection by mimicking benign behaviors. Besides that, the malware is engineered to unlock its malicious payload when it reaches a target (endpoint) with a loose predefined set of attributes. In the study, those attributes were the biometrics feature of the target such as facial and voice features. The malware uses CNN to detect and confirm target identity, and upon target confirmation, an encryption key is generated and used by the WannCry malware to encrypt the files on the target endpoint device. The encryption key is only generated by matching the voice and the facial features of the target. This means reverse engineering the malware using static analysis is not useful to recover the encryption key.

2.4 Practical Challenges

The new and emerging malware threats discussed in section 2.3 provide strong evidence for the need of adopting dynamic and behavioural analysis to build malware detection tools. The use of machine learning is the most promising technique to implement malware detectors and tools that apply behavioural analysis as shown in section 2.2. While the use of machine learning for malware detection has shown promising results in both static and dynamic analysis, there are significant challenges that limit the success of machine learning based malware detectors in the wild.

2.4.1 Cost of Training Detectors

The first challenge is the cost of training and updating malware detectors in production environments. Malware detection is unlike other domains where machine learning techniques have been applied successfully such as computer vision, natural language processing, and e-commerce. Malware instances evolve and change their behaviors over a short period; some studies by antimalware vendors reported that a new malware instance could change its behaviors in less than 24 hours since it has been released [13, 2]. This means a frequently trained machine learning model will become out-dated. This also means we need to frequently retrain our malware detectors to be able to detect new and mutated malware instances. Therefore, adaptability in machine learning models for malware detection is a crucial requirement and not just an ancillary capability.

Recently, the challenge of adaptability, and scalability of machine learning models for malware detection in the wild has become obvious [25]. The majority of the work proposed in the literature has done very little to reduce and optimize the feature space to design detectors ready for early malware detection in a production environment [14]. For instance, it is not clear how the proposed detection methods will scale when the number of monitored endpoints increases. Unlike computer vision, natural language processing and other areas that utilize machine learning, malware instances continue to evolve and change. This mostly requires retraining machine learning mod-

els in production, which is an expensive and complicated task. Therefore, when using machine learning for malware detection, we need to think differently. New methods to reduce the cost of retraining malware detectors and improve detection quality are urgent.

2.4.2 Malware Detector Interpretability

Cybersecurity analysts always prefer solutions that are interpretable and understandable, such as rule-based or signature-based detection. This is because of the need to tune and optimize these solutions to mitigate and control the effect of false positives and false negatives. Interpreting machine learning models is a new and open challenge [32]. However, it is expected that an interpretable machine learning solution will be domain specific, for instance, interpretable solutions for machine learning models in healthcare are different than solutions in malware detection [1]. Any malware detector will generate false positives, and unless malware analysts can understand and interpret the reason that a benign application was wrongly classified as malicious, they will not accept those black box malware detectors. To our knowledge, no work in the literature investigated the interpretability of machine learning models for malware detection.

One difficulty with machine learning interpretability in this domain is that many of the features are not meaningful to humans without the context which they appear in. This means it is necessary to map back from features to raw data in order to better understand the feature and its context at the time of classification. The problem of mapping features to raw data is touched upon in Chapters 4 and 5 and applies to malware detectors which use ngrams or binaries converted to images. However, the techniques used to map features to raw data will also be application specific.

2.4.3 Adversarial Malware

Last but not least, a malware detection system utilizing machine learning could be defeated using adversarial malware samples. For instance, Kolosnjaji et al. showed

in [18] that by using an intelligent evasion attack they can defeat the deep learning detection system proposed in [29] by Raff et al. They simply used their knowledge of how the proposed deep learning detection system operates and designed a gradient-based attack as an evasion technique to overcome it. With adversarial malware, the system detection accuracy dropped from 94.0% to almost 50.0%. Machine learning algorithms are not designed to work with adversarial examples. Grosse et al. demonstrated that using adversarial malware samples; they could reduce the detection accuracy of a malware detection system that uses static analysis and machine learning to 63.0% [12]. They also showed that adopting anti adversarial machine learning techniques used in computer vision is not effective in malware detection. Yang et al. proposed adversarial training as a solution for adversarial malware [37]. They designed a method for adversarial android malware instances generation. The proposed method requires access to the malware binaries and source code, besides, it is mainly useful for static malware detection systems.

2.5 Bridging the Detection Gap

To overcome the challenges we discussed in section 2.4, we propose new solutions to mitigate these challenges and reduce the gap.

2.5.1 Disposable Micro Detectors

Current best practices in constructing and building machine learning models follow a monolithic architecture. In a monolithic architecture, a single computationally expensive (to build and train) machine learning model is used to detect malware. While this architecture or approach for building machine learning models is successful in other domains, we believe it is unsuitable for malware detection given the highly evolving characteristics of malware instances. We propose a new approach inspired by the microservices architecture. In this approach, multiple, small, inexpensive, focused machine learning models are built and orchestrated to detect malware instances. Each model or detector is built to detect the behaviors of a specific malware instance (e.g.,

Mirai, WannaCry), or at most a single malware family (a group of similar malware instances). Also, each model or detector is built using features that are similar, such as having the same computational cost, or unique to the specific execution environment. This is because out of the super set of features designed to detect malware, it is common that a subset of these features could be more or less useful to detect a specific malware instance or family. The use of micro (small) and focused detectors reduce the cost of retraining and deployment in production. This is because detectors for new malware could be trained and added without the need to retrain existing detectors. In addition, when malware detectors become outdated as a result of a malware's evolving behavior, the outdated detectors are disposed of and replaced by new ones. The use of micro-detectors enables adaptability by design rather than attempting to change machine learning models and algorithms to support adaptability.

2.5.2 Analyst Friendly Interpretation

Adopting sophisticated machine learning techniques for malware detection in a production environment is a challenge. This is because most of the time it is not possible to understand how the machine learning systems make their malware detection decisions. Therefore, tuning and maintaining these systems is a challenge in production and new techniques for malware analysts to interpret and evaluate the performance of malware detectors are needed. We propose the use of evolutionary computation techniques such as genetic algorithms or clonal selection algorithms to generate an interpretation for black-box machine learning models such as deep learning. Using evolutionary computation, we could describe the decisions of malware detectors using a set of IF-Then rules. The only information required is the input features the malware detector uses to make a decision.

The IF-Then rules are useful to explain the behaviors that trigger a specific decision (e.g., malicious or benign) by the malware detector. Cybersecurity and malware analyst are comfortable working with IF-Then rules. These rules will help in understanding the decision made by malware detectors, explain the scope of the detection, and identify potential over generalization or overfitting that could result in false pos-

itives or false negatives.

It is essential that the IF-Then rules set interpretation of the malware detector is expressed in raw malware behaviors and not in machine learning features. Machine learning features are most likely understandable by machine learning engineers and experts. The interpretation should be acceptable to a malware analyst who does not need to be a machine learning expert.

Other model specific Interpretations techniques, such as the ones discussed in chapters 4 and 5 of this thesis, can be utilized to improve model confidence so stakeholders are more likely to trust machine learning based malware detectors. Further, these approaches can ensure the model is not easily manipulated and thus prone to adversarial malware by over-relying on easy to change and superficial features. In this way, interpretation can help ensure model *robustness*. Lastly, machine learning models learn complex patterns that can be utilized making beyond making classification decisions. Interpretation can be used so that patterns learn by a machine learning model can help malware analyst with downstream tasks such as finding import snippets of code.

2.5.3 Anti Adversarial Malware

To improve the resilience of malware detectors against adversarial malware, we believe it is essential to study the effort required by the malware authors to design an adversarial malware for specific malware detectors. For example, what technique a malware author would use to probe and study a malware detector in production to design a malware that could bypass a detector.

Measuring the effort to probe detectors and design adversarial malware under two main settings is essential. The first setting is black-box, where the malware authors have minimum knowledge about the malware detector's internal design and the features used by the machine learning algorithm. The second setting is white-box, where the malware authors have sufficient knowledge about the malware detector's internal design and the machine learning algorithm. Training and updating malware detectors is likely the most efficient solution against adversarial malware. Knowing

the effort needed to evade a malware detector will help in designing training strategies and policies to increase the effort required to evade the detectors.

As we mentioned before, Cohen provided a formal proof that creating a perfect malware detection system is not possible [7, 6]. We believe that designing a perfect adversarial malware is not possible. Therefore we expect that using ensemble-based hybrid machine learning approach for malware detectors will be effective against adversarial malware. It is expected that by creating a malware detector using an ensemble hybrid machine-learning approach, the risk of evading detection will decrease and the effort to design adversarial malware will increase. A hybrid machine learning model is when two or more different machine learning algorithms are used to construct the model. In the literature, adversarial malware samples evade malware detectors that use a single machine learning algorithm or technique [37, 12, 18]. In our method, a hybrid machine learning approach for building a malware detector is an approach to provide a defense-in-depth model for malware detectors.

2.6 Conclusion

In this Chapter, we reviewed the current state-of-the-art in malware detection using machine learning. We discussed the recent trends in malware development and emerging malware threats. We argued that behavioral analysis would dominate the next generation anti malware systems. We discussed the challenges of applying machine learning to detect malware in the wild and proposed our thoughts on how we could overcome these challenges. Machine learning malware detectors require inexpensive training methods; they need to be interpretable for the malware analysts and not only for machine learning experts. Finally, they need to tolerate adversarial malware by design

References

[1] M. A. Ahmad, A. Teredesai, and C. Eckert. "Interpretable Machine Learning in Healthcare". In: *2018 IEEE International Conference on Healthcare Informatics (ICHI)*. June 2018, pp. 447–447. DOI: 10.1109/ICHI.2018.00095.

[2] K. Allix et al. "Are Your Training Datasets Yet Relevant?" In: *Engineering Secure Software and Systems*. Ed. by Frank Piessens, Juan Caballero, and Nataliia Bielova. Cham: Springer International Publishing, 2015, pp. 51–67. ISBN: 978-3-319-15618-7.

[3] A. Azmoodeh et al. "Detecting crypto-ransomware in IoT networks based on energy consumption footprint". In: *Journal of Ambient Intelligence and Humanized Computing* 9.4 (Aug. 2018), pp. 1141–1152. ISSN: 1868-5145. DOI: 10.1007/s12652-017-0558-5. URL: https://doi.org/10.1007/s12652-017-0558-5.

[4] D. M. Chess and S. R. White. "An Undetectable Computer Virus". In: *In Proceedings of Virus Bulletin Conference* (2000).

[5] Y. Han Choi et al. "Toward extracting malware features for classification using static and dynamic analysis". In: *2012 8th International Conference on Computing and Networking Technology (INC, ICCIS and ICMIC)*. Aug. 2012, pp. 126–129.

[6] F. Cohen. "Computational aspects of computer viruses". In: *Computers & Security* 8.4 (1989), pp. 297–298.

[7] F. Cohen. "Computer viruses: Theory and experiments". In: *Computers & Security* 6.1 (1987), pp. 22–35.

[8] J. Jang D. Kirat and M. Stoecklin. *DeepLocker Concealing Targeted Attacks with AI Locksmithing*. Aug. 2018.

[9] O. E. David and N. S. Netanyahu. "DeepSign: Deep learning for automatic malware signature generation and classification". In: *2015 International Joint*

Conference on Neural Networks (IJCNN). July 2015, pp. 1–8. DOI: 10.1109/IJCNN.2015.7280815.

[10] D. de Dréziguć, J.P. Fizaine, and N. Hansma. "In-depth analysis of the viral threats with OpenOffice.org documents". In: *Journal in Computer Virology* 2.3 (Dec. 2006), pp. 187–210. URL: https://doi.org/10.1007/s11416-006-0020-2.

[11] Global Research and Analysis Team, KASPERSKY Lab. *Fileless Attack Against Enterprise Network*. White Paper. KASPERSKY Lab, 2017.

[12] K. Grosse et al. "Adversarial Examples for Malware Detection". In: *Computer Security – ESORICS 2017*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Cham: Springer International Publishing, 2017, pp. 62–79. ISBN: 978-3-319-66399-9.

[13] A. Gupta et al. "An empirical study of malware evolution". In: *2009 First International Communication Systems and Networks and Workshops*. Jan. 2009, pp. 1–10. DOI: 10.1109/COMSNETS.2009.4808876.

[14] G. Hajmasan, A. Mondoc, and O. Creţ. "Dynamic behavior evaluation for malware detection". In: *2017 5th International Symposium on Digital Forensic and Security (ISDFS)*. Apr. 2017, pp. 1–6. DOI: 10.1109/ISDFS.2017.7916495.

[15] M. Hassen, M. M. Carvalho, and P. K. Chan. "Malware classification using static analysis based features". In: *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*. Nov. 2017, pp. 1–7.

[16] C. Karam and V. Kamluk. "Blockchainware - Decentralized Malware on The Blockchain". In: *Black Hat ASIA*. 2015.

[17] S. Kilgallon, L. De La Rosa, and J. Cavazos. "Improving the effectiveness and efficiency of dynamic malware analysis with machine learning". In: *2017 Resilience Week (RWS)*. Sept. 2017, pp. 30–36.

[18] B. Kolosnjaji et al. "Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables". In: *CoRR* abs/1803.04173 (2018). arXiv: 1803.04173. URL: http://arxiv.org/abs/1803.04173.

[19] H. Lim et al. "Malware classification method based on sequence of traffic flow". In: *2015 International Conference on Information Systems Security and Privacy (ICISSP)*. Feb. 2015, pp. 1–8.

[20] A. Magnusardottir. *Fileless Ransomware: How It Works & How To Stop it?* White Paper. Cyren, June 2018.

[21] F. Martinelli et al. "I find your behavior disturbing: Static and dynamic app behavioral analysis for detection of Android malware". In: *2016 14th Annual Conference on Privacy, Security and Trust (PST)*. Dec. 2016, pp. 129–136.

[22] C. Miller and C. Valasek. *Remote Exploitation of an Unaltered Passenger Vehicle*. White Paper. Black Hat, Oct. 2015.

[23] J. Moubarak, M. Chamoun, and E. Filiol. "Developing a K-ary malware using blockchain". In: *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. Apr. 2018, pp. 1–4. DOI: 10.1109/NOMS.2018.8406331.

[24] H. Naeem, B. Guo, and M. R. Naeem. "A light-weight malware static visual analysis for IoT infrastructure". In: *2018 International Conference on Artificial Intelligence and Big Data (ICAIBD)*. May 2018, pp. 240–244.

[25] A. Narayanan et al. "Adaptive and scalable Android malware detection through online learning". In: *2016 International Joint Conference on Neural Networks (IJCNN)*. July 2016, pp. 2484–2491. DOI: 10.1109/IJCNN.2016.7727508.

[26] D. OSahın et al. "New results on permission based static analysis for Android malware". In: *2018 6th International Symposium on Digital Forensic and Security (ISDFS)*. Mar. 2018, pp. 1–4.

[27] A. De Paola et al. "A hybrid system for malware detection on big data". In: *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. Apr. 2018, pp. 45–50.

[28] A. O. Prokofiev, Y. S. Smirnova, and V. A. Surov. "A method to detect Internet of Things botnets". In: *2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*. Jan. 2018, pp. 105–108.

[29] E. Raff et al. "Malware Detection by Eating a Whole EXE". In: *CoRR* abs/1710.09435 (2017).

[30] M. Rigaki and S. Garcia. "Bringing a GAN to a Knife-Fight: Adapting Malware Communication to Avoid Detection". In: *2018 IEEE Security and Privacy Workshops (SPW)*. May 2018, pp. 70–75.

[31] A. A. Selcuk, F. Orhan, and B. Batur. "Undecidable problems in malware analysis". In: *2017 12th International Conference for Internet Technology and Secured Transactions (ICITST)*. 2017, pp. 494–497.

[32] S. Shirataki and S. Yamaguchi. "A study on interpretability of decision of machine learning". In: *2017 IEEE International Conference on Big Data (Big Data)*. Dec. 2017, pp. 4830–4831. DOI: 10.1109/BigData.2017.8258557.

[33] S. W. Soliman, M. A. Sobh, and A. M. Bahaa-Eldin. "Taxonomy of malware analysis in the IoT". In: *2017 12th International Conference on Computer Engineering and Systems (ICCES)*. Dec. 2017, pp. 519–529.

[34] J. Su et al. "Lightweight Classification of IoT Malware Based on Image Recognition". In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 01. July 2018, pp. 664–669.

[35] M. Su and K. Fung. "Detection of android malware by static analysis on permissions and sensitive functions". In: *2016 Eighth International Conference on Ubiquitous and Future Networks (ICUFN)*. July 2016, pp. 873–875.

[36] *The 2017 State of Endpoint Security Risk*. White Paper. Ponemon Institute, 2017.

[37] W. Yang et al. "Malware Detection in Adversarial Settings: Exploiting Feature Evolutions and Confusions in Android Apps". In: *ACSAC*. 2017.

[38] M. Yeo et al. "Flow-based malware detection using convolutional neural network". In: *2018 International Conference on Information Networking (ICOIN)*. Jan. 2018, pp. 910–913. DOI: 10.1109/ICOIN.2018.8343255.

[39] L. Zhang-Kennedy et al. "The Aftermath of a Crypto-ransomware Attack at a Large Academic Institution". In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Baltimore, MD, USA: USENIX Association, 2018, pp. 1061–1078. ISBN: 978-1-931971-46-1. URL: http://dl.acm.org/citation.cfm?id=3277203.3277282.

CHAPTER 3

JSLess: A Tale of Fileless JavaScript Memory-Resident Malware

Sherif Saad, Farhan Mahmood, and William Briguglio

In Proceedings of the 15th International Conference on Information Security Practice and Experience

3.1 Introduction

Fileless malware is a new class of the memory-resident malware family that successfully infects and compromises a target system without leaving a trace on the target filesystem or secondary memory (e.g., hard drive). Fileless malware infects the target's main-memory (RAM) and executes its malicious payload. Fileless malware is not just another memory-resident malware. To our knowledge, Fred Cohen developed the first memory-resident malware (Lehigh Virus) in the early 80s. This usually leads some researchers to believe that fileless malware is not a new malware threat but only a new name for an old threat. However, this is not true, fileless malware has some distinguishing properties. First, malware attacks require some file infection or writing to the hard drive, this includes traditional memory resident malware. Fileless malware infection and propagation does not require writing any data to the target device filesystem. However, it is possible that the malicious payload (e.g., the end goal) of the fileless malware writes data to the hard drive, for example, a fileless ransomware, but again the ransomware propagation and infection are fileless. The

second key property of fileless malware is that it depends heavily on using benign software utilities and libraries already installed on the target device to execute the malicious payload. For instance, a fileless ransomware will use cryptographic libraries and APIs already installed on the target to complete its attack rather than installing new cryptographic libraries or implementing its own.

There are other unique properties of fileless malware, but the most important ones are the fileless infection approach and the use of benign utilities and libraries of the compromised machine to execute the malicious payload. Those two properties of fileless malware make it an effective threat in evading and bypassing sophisticated anti-malware detection systems. This is because most anti-malware relies on scanning the compromised filesystem to detect malware infections. Also, because fileless malware use legitimate software utilities and programs to attack computer systems, it is challenging for anti-malware systems that use dynamic analysis to detect fileless malware. Moreover, being fileless is an anti-forensics technique, since it does not leave any trace after the attack is complete, it is tough for forensics investigator to reverse engineer the malware.

Fileless malware attacks and incidents are already observed in the wild compromising large enterprises. According to KASPERSKY lab, 140 enterprises were attacked in 2017 using fileless malwares [5]. Ponemon Institute reported that 77% of the attacks against companies use fileless techniques [18]. Also, CYREN recently reported that during 2017 there was over 300% increase in the use of fileless attacks. Moreover, they expected that the new generation of Ransomware would be fileless [7]. This expectation proved to be correct when TrendMicro reported the analysis of SOREBRECT Ransomware, the first fileless ransomware attack in the wild [19]. However, we think that it is inaccurate to describe SOREBRECT Ransomware as fileless malware, since it places an executable file on the compromised machine which injects the malicious payload into a running system process. Then, it deletes the file and any trace on the system logs using a self-destruct routine. Because the infection and the injection of SOREBRECT Ransomware requires placing files on the compromised host, we do not think it is a true fileless malware. Moreover, deleting the files is not enough to

hide the trace, file carving techniques could be used to recover the deleted files.

Another common trend in developing fileless malware is the use of Microsoft PowerShell. PowerShell is a command-line shell and scripting language that allows system administrators to manage and automate tasks related to running processes, the operating system, and networks. It is preinstalled by default on new Windows versions and it can be installed on Linux and MacOS systems. PowerShell is a good example of a benign and powerful system utility that could be used by fileless malware. Several reports by anti-malware vendors discuss how malware authors take advantages of PowerShell to develop sophisticated fileless malware [10].

In this chapter, we summarize our research on fileless malware attacks in modern web applications. We investigate the possibility of developing a fileless malware using modern JavaScript(JS) features that were introduced with HTML5. In our assessment of the potential threats of fileless malware attacks, we explore the use of benign JavaScript and HTML5 features to develop fileless malware. Based on our analysis we implemented **JSLess** as a proof-of-concept(PoC) fileless JavaScript malware that successfully infects a web browser and executes several malicious payloads.

The contribution of this Chapter is threefold. First, identify the malicious potential of new benign features in web technology and how they could be used to develop fileless malware. Second, design and implement JSLess as a PoC fileless JS malware that uses a new dynamic injection method and advanced evasion techniques to infect modern web apps and execute a variety of attacks. Third, demonstrate the threats of fileless malware in modern web applications by evaluating the proposed fileless malware with several free and commercial malware detection tools that apply both static and dynamic analysis.

This chapter is organized as follows; section 3.2 is a literature review of fileless malware and JavaScript malware. In section 3.3, we explain new benign features in modern JavaScript and HTML5 and their security issues. Then, in section 3.4 we present our JavaScript fileless malware design and implementation. Next, in section 3.5 we evaluate the evasion behaviors of the JS fileless malware against free and commercial anti-malware tools, then we discuss possible detection and mitigation

techniques. Finally, a conclusion and possible future work is presented in section 3.6.

3.2 Literature Review

Code injection attacks have been studied from different perspectives in the literature. The research in this area tried to detect malicious behaviors in JavaScripts using various methods, including signature-based analysis, utilizing machine learning algorithms, using honeynets, and applying several deobfuscation techniques. This section discusses the main research directions in this area and highlights some of the most important contributions in the literature.

S. Yoon et al. proposed a method to generate unique signatures for malicious JavaScripts [23]. The authors used content-based signature generation techniques and utilized the Term Frequency - Inverse Document Frequency (TF-IDF) and Balanced Iterative Reducing and Clustering using Hierarchies methods to generate the conjunction signatures for JavaScripts [23]. Although signature-based analysis can help detect several malicious behaviours, the work in [23] is based on the assumption that the attack type of the input JavaScripts is known, which is not always a practical assumption in real-life environments. Moreover, obfuscation remains a challenging problem that reduces the effectiveness of signature-based techniques.

G. Blanc et al. tried to address the obfuscation problem by applying abstract syntax tree (AST) based methods to characterize obfuscating transformations found in malicious JavaScript [2]. The authors used AST-based methods to demonstrate significant regularities in obfuscated JavaScript programs. The work in [2] is based on generating AST fingerprints (ASTFs) for each JS file present in their learning dataset then manually picking representative subtrees for further processing. The manual intervention in this procedure and relying only on the training data sets without providing a mechanism to update the training set with new samples raise many questions about the feasibility of this solution. Moreover, the work in [2] did not consider the different categories of obfuscation techniques in real-world malicious JavaScript, which was analyzed by W. Xu et al. in [22]. Similar work was done by I.

AL-Taharwa1 et al. to detect obfuscation in JavaScript using semantic-based analysis based on the variable length context-based feature extraction (VCLFE) scheme that takes advantage of AST representation [17].

One controversial issue in this area of research is the physical location where the detection mechanism takes place. One approach is to collect and analyze HTTP traffic via local proxy and implement the detection algorithm on the proxy side [12]. Another approach is to implement the detection mechanism on the client side, such as the work done by V. Sachin et al., who used light-weight JavaScript instrumentation that enables static and dynamic analysis of the visited webpage to detect malicious behavior [13]. R. K. Kishor et al. took an extra step and developed an extension that can be installed on the client web browser to detect malicious web contents [6]. Similar work was done by C. Wang et al., who focused on the browser detection mechanism integrated with HTML5 and Cross Origin resource sharing (CORS) properties [20].

In recent years, JavaScript became a very popular solution for hybrid mobile applications. This recent adoption of technology in mobile applications poses a new risk of malicious code injection attacks on mobile devices. J. Mao et al. proposed a method to detect anomalous behaviors in hybrid Android apps as anomalies in function call behaviors [9]. The authors instrumented the JavaScript code dynamically in the JavaScript engine to intercept function calls of JavaScript in hybrid apps. They also extracted events from the Android WebView component to enhance the performance of their proposed detection model [9].

Since the feature engineering step is the core of any machine-learning malware detection solution, many researchers focused on developing a feature engineering methodology. H. Adas et al. proposed a method to extract inspection features from over two million mobile URLs [1]. The authors used a MapReduce/Hadoop based cloud computing platform to train and implement their classifier and evaluate its performance. Although this is a good step towards building a cloud-based classifier, more experiments need to be conducted to evaluate its efficiency with respect to real-time detection of malware. Moreover, the classification model in [1] was trained with features based on the static analysis of the malicious code, which is not an efficient

approach in detecting most fileless malwares.

S. Ndichu et al. developed a neural network model that can be trained to learn the context information of texts [11]. The main contribution of the work in [11] is developing a new feature extraction method and using unsupervised learning algorithms that produce vectors of fixed lengths. These vectors can be used to train a neural network that classifies the JavaScript code as normal or malicious [11]. Similar work was done earlier by Y. Wang et al. using deep learning [21]. Wang et al. used deep features extracted by stacked denoising auto-encoders (SdA) to detect malicious JavaScript codes [21].

Neural networks were not the only machine learning framework used to detect malicious JavaScript codes. Seshagiri et al. used Support Vector Machine (SVM) to detect malicious JavaScript codes [15]. Features were extracted using static analysis of web pages. Although ML is a promising solution, there are many challenges that face developers during the implementation of such solutions. The main challenge is creating a feature vector that can truly characterize the behaviour of fileless malware. Fileless malware does not leave clear traces on the victim's machine and therefore are very difficult to identify.

Other research directions are considered in the literature. The following are few examples of different approaches considered by researchers in the last few years. B. Sayed et al. proposed a model that uses information flow control dynamically at run-time to detect malicious JavaScript [14]. Y. Fange et al. used Long Short-Term Memory (LSTM) to develop a malicious JavaScript detection model [4]. V. Shen used a high-level fuzzy Petri net (HLFPN) to detect JavaScript malware [16]. D. Cosovan used hidden markov models and linear classifiers to detect JavaScript-based malware [3]. Last but not least, D. Maiorca et al. used discriminant and adversary-aware API analysis to detect malicious scripting code[8].

Although the previous work in this research area presented promising results, there are many challenges that prevent accurate detection of fileless malwares in real web applications. To highlight the significance of the threat posed by fileless malwares, this chapter presents a practical design and implementation of a fileless malware as

a PoC to demonstrate the threats of fileless malware in web applications.

3.3 Benign Features with Malicious Potentials

With the introduction of HTML5, a new generation of modern web applications become a reality. This is mainly because HTML5 introduced a rich-set of powerful APIs and features that can be used by JavaScript. Some of the new features and APIs in HTML focus on enabling the development of web apps with high connectivity and performance. Further, HTML5 provides a set of APIs that allow web applications written in JavaScript to access information about the host running the web app and also other peripheral devices connected to the host. For instance, a web app developed with HTML5 and JavaScript could have access to the user geolocation, device orientation, mic, and camera.

While these new powerful features were proposed to improve web application development, we found in our analysis of these features that hackers and malware authors could misuse them. Many of these benign features have serious malicious potential. In this section, we will mainly focus on HTML5 features that were proposed to boost web application performance, scalability, and connectivity.

3.3.1 WebSockets

WebSocket is a new communication protocol that enables a web-client and a web-server to establish a two-way (full-duplex) interactive communication channel over a single TCP connection. It provides bi-directional real-time communication which is an urgent requirement for modern interactive web applications. With WebSocket, the communication method between the web-client and the web-server is not limited to pull-communication. Instead, push-communication and even an interactive communication become possible. For this reason, WebSocket becomes the dominate technology in developing instant messaging apps, gaming applications, streaming services, or any web app which requires data exchange between the client and the server in real-time.

WebSocket is currently supported by all major web browsers such as Chrome, Firefox, Safari, Edge, and IE. Moreover, the WebSocket protocol is supported by common programming languages such as Java, Python, C#, and others. This enables the development of desktop, mobile apps, or even microservices that communicate using WebSocket as a modern and convenient communication protocol.

It is clear that by using WebSocket the connectivity of web apps becomes much higher quality and much more reliable. However, WebSocket is considered by web security researchers a security risk. WebSocket enables a new attack vector for malicious actors. Common web attacks such as cross-site scripting (XSS) and man in the middle (MitM) are possible over WebSockets. WebSocket by design does not obey the same-origin policy; this means the web browser will allow a WebSocket script to connect to different web pages even if they do not share the same-origin (same URI scheme, host and port number). Again WebSocket by design is not bound by cross-origin resource sharing (CORS). This means a web app running inside the client web browser could request resources that have a different origin from the web app. This flexibility could be easily abused by malicious actors as we will demonstrate in the next section.

3.3.2 WebWorker

Originally JavaScript is a single-threaded language which means in any web app there is only a single line of code or statement that can be executed at any given time. As a result, JavaScript cannot perform multiple tasks simultaneously. WebWorker is a new JavaScript feature that was introduced with HTML5 to improve the performance of the JavaScript applications. WebWorker enables JavaScript code to run in a background thread separate from the main execution thread of a web app. In other words WebWorker allows web applications to execute tasks in the background without impacting the user interface as it works completely separate from the UI thread. For this reason, WebWorkers are typically used to run long and expensive operations without blocking the UI. For instance, the code in listing 3.1 initializes a new web worker object and runs the code in worker.js asynchronously in a new thread.

```
if (typeof(worker) == "undefined") {
    worker = new Worker("worker.js");
}
```

Listing 3.1: WebWorker Initialization Example

WebWorker should be used to do computationally intensive tasks to avoid blocking the UI or any other code executed in the main thread. If a computationally intensive task executes in the main JavaScript thread, the web app will freeze and become unresponsive to the user. WebWorker is currently supported by all major web browsers such as Chrome, Firefox, Safari, Edge, and IE.

As we can see WebWorker is an essential feature for developing a modern and responsive web application. However, the devil is in the details. While WebWorker seems like a harmless feature, it opens the door for several malicious scenarios and security issues. For example, is allows DOM-based XSS. CORS does not bind it, and hence a web worker could share and access resources from different origins. But in our opinion, the most critical security issue with WebWorker is its ability to insert silent running JavaScript code. This could enable a malicious payload to run in a background thread created by malicious or compromised web apps. One possible example is using a WebWorker with a malicious web app to preform cryptocurrency mining without the users' consent. On the bright side, the WebWorker will terminate if the user closes the web browser or the web app that created the web worker object. However, as we will see in the next subsection, malware authors can work around this with ServiceWorkers.

3.3.3 ServiceWorker

ServiceWorker is another new appealing JavaScript feature. We could consider ServiceWorker as a special type of WebWoker. ServiceWorker allows running JavaScript code in a separate background thread. This is very similar to WebWorker but unlike WebWorker, the lifetime of the ServiceWorker is not tied to a specific webpage or even the web browser. This means even if the user navigates away from the web

app that created the ServiceWorker or closes the web browser, the ServiceWorker will continue to run in the background. The ServiceWorker will normally terminate when it has complete its task (e.g., execute its script) or received a termination signal from the web server, or terminate abnormally as a result of a crash, system reboot or shutdown.

ServiceWorker was introduced to enable a rich offline experience to users and improve the performance of modern web apps. The code in listing 3.2 shows an example that creates a ServiceWorker from the file sw_demo.js. ServiceWorkers share the same security issues and risks that exist in WebWorkers but the lifetime of the security risks persists longer.

```
window.addEventListener('load', () => {
  navigator.serviceWorker.register('/sw_demo.js')
  .then((registration) => {
    // ServiceWorker registered successfully
  }, (err) => {
    // ServiceWorker registration failed
  });
});
```

Listing 3.2: ServiceWorker Registration Example

3.4 JavaScript Fileless Malware

In this section, we explain how the benign JavaScript features we introduced in section 3.3 could be used to implement a fileless JavaScript malware. To demonstrate this threat, we designed and implemented JSLess as a PoC fileless malware. We designed JSLess as a fileless polymorphic malware, with a dynamic malicious payload, that applies both timing and event-based evasion.

3.4.1 Infection Scenarios

In our investigation, we define two main infection scenarios. The first scenario is when the victim (web user) visits a malicious web server or application as illustrated

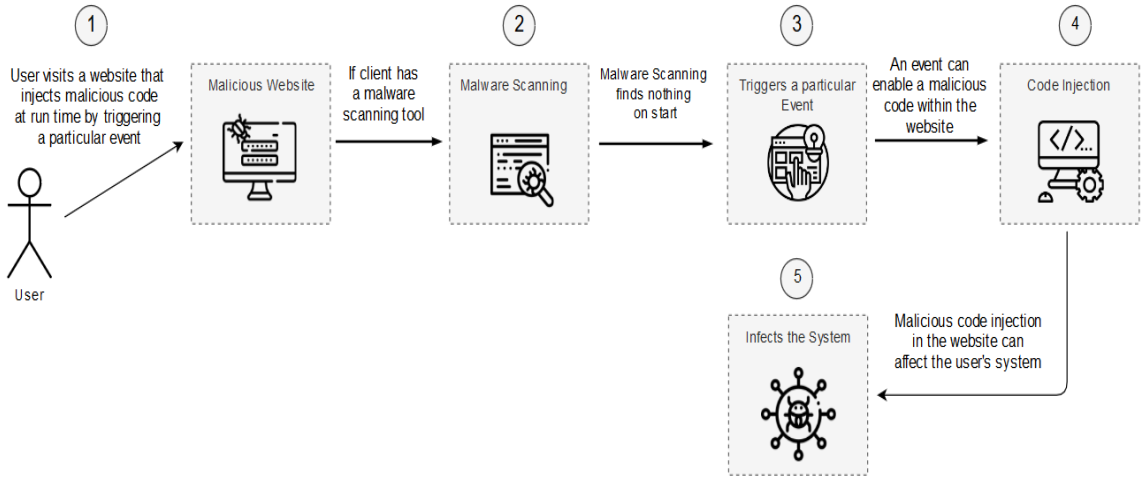

Fig. 3.4.1: JavaScript Fileless Malware First Infection Scenario

in figure 3.4.1. In this case, the malicious web server will not show any malicious behaviors until a specific event triggers the malicious behavior. In our demo, the attack posts specific text messages on a common chat room. The message act as an activation command to the malware. When the message is received the malware is injected dynamically into the victim's browser and starts running as part of the script belonging to the public chat room.

The second infection scenario is when the malware compromises a legitimate web application or server to infect the web browsers of the users who are currently visiting the compromised website as illustrated in figure 3.4.2. In this case, both the website and the website visitors are victims of the malware attack. The malware will open a connection with the malicious server (e.g., C&C server) that hosts the malware to download the malicious payload or receive a command from the malware authors to execute on the victim browser.

Note that in both scenarios the malicious code infection/injection happens on the client side, not the server side.

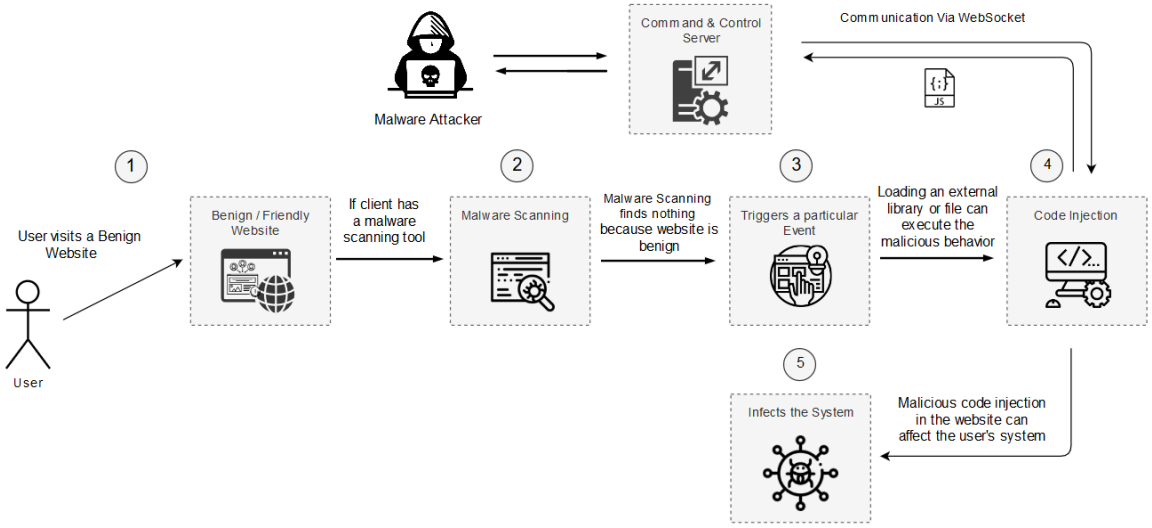

Fig. 3.4.2: JavaScript Fileless Malware Second Infection Scenario

3.4.2 Operational Scenarios

JSLess is delivered to the victim's web browser through a WebSocket connection. When the victim visits a malicious web server, the WebSocket connection will be part of the web app on the malicious server. However, if the malware authors prefer to deliver JSLess by compromising a legitimate web app/server to increase in the infection rate, then the WebSocket delivery code could be added into a third-party JavaScript library (e.g. JQuery). Almost all modern web application relies on integrating third-party JavaScript files. The WebSocket delivery code is relatively simple (see the code in listing 3.3) and could easily be hidden in a malicious third-party script library that is disguised as legitimate. Alternatively, the code could be inserted via an HTML injection attack on a vulnerable site that does not correctly sanitize the user input.

```
MalWS = new WebSocket('{{WSSurl}}/KeyCookieLog.js');
MalWS.onmessage = function(e) {
    sc = document.createElement('script');
    sc.type = 'text/JavaScript';
    sc.id = 'MalSocket';
    sc.appendChild(document.createTextNode(e.data));
    B = document.getElementsByTagName("body");
    B[0].appendChild(sc);
};
```

Listing 3.3: malicious payload delivered with websocket

The WebSocket API is used to deliver the malware source code in JavaScript to the victim browser. Once the connection is opened, it downloads the JavaScript code and uses it to create a new script element which is appended as a child to the HTML file's body element. This causes the downloaded script to be executed by the client's web browser.

Delivering the malware payload over WebSocket and dynamically injecting it into the client's web browser provides several advantages to malware authors. The fact that the malware code is only observable when the web browser is executing the code and mainly as a result of a trigger event provides one important fileless behavior for the malware. The malicious code is never written to the victim's file system. Using WebSocket to deliver the malware payload does not raise any red flags by anti-malware systems since it is a popular and common benign feature. Using benign APIs is another essential characteristic of fileless malware.

The fact that JSLess can send any malicious payload for many attack vectors and inject arbitrary JavaScript code with the option to obfuscate the injected malicious code enables the design of polymorphic malware. All of these attributes make JSLess a powerful malware threat that can easily evade detection by anti-malware systems. For instance, a pure JavaScript logger could be quickly injected in the user's browser to captures user's keystroke events and send them to the malware C&C server over WebSocket. Note that benign and native JavaScript keystroke capturing APIs are used which again will not raise any red flags. Figure 3.4.3 shows an example of an

Fig. 3.4.3: Obfuscated JavaScript code injection

injected obfuscated JavaScript key logger that captures keystroke events and sends it to the malware C&C server over WebSockect.

To utilize the victim's system's computation power or run the malicious scripts in a separate thread from the main UI thread, JSless takes advantage of WebWorkers. This allows JSless to run malicious activities that are computationally intensive, such as cryptocurrency mining. The WebWorker script is downloaded from the C&C server. The JavaScript code in listing 3.4 shows how the malicious WebWorker code could be obtained as a blob object and initiated on the victim's browser. Using the importScripts and createObjectURL functions, we were able to load a script from a different domain hosted on the different server and execute it in the background of the benign web app.

```
blob = new Blob(["self.importScripts('{{HTTPSurl}}/foo.js');"],
       {type: 'application/JavaScript'});

w = new Worker(URL.createObjectURL(blob));
```

Listing 3.4: Breaking Same-origin Policy with ImportScripts()

Until this point one limitation of JSless malware-framework is that fact that the malware will terminate as soon as the user closes his web browser or navigates away

from the compromised/malicious web server. This limitation is not specific to JSless, it is the common behavior of any fileless malware. In fact, many malware authors sacrifice the persistence of their malware infection by using fileless malware to avoid detection and bypass anti-malware systems. However, that does not mean fileless malware authors are not trying to come up with new methods and techniques to make their fileless malware persistent. In our investigation to provide persistence for JSless even if the user navigates away from the compromised/malicious web page or closes the web browser. We take advantage of the ServiceWorker API to implement a malware persistence technique with minimal footprint.

To achieve malware persistence, we used the WebSocket API to download a script from the malicious server. After downloading the ServiceWorker registration code from the malicious server, as shown in listing 3.1, it registers a sync event, as shown in listing 3.5, to cause the downloaded code to execute and stay alive even if the user has navigated away from the original page or closed the web browser. The malicious code will continue to run and terminate normally when it is completed or abnormally as result of exception, crash, or if the user restarts his machine. Note that when we use ServiceWorker, a file is created and temporarily stored on the client machine while the ServiceWorker is running. This is the only case where JSless will place a file on the victim machine, and it is only needed for malware persistence.

```
self.addEventListener('sync', function (event) {
 if (event.tag === 'mal-service-worker') {
  event.waitUntil(malServiceWorker()
  .then((response) => {
    // Service Worker task is done
  }));
 }
});

function malServiceWorker() {
 // Malicious activity can be performed here
}
```

Listing 3.5: ServiceWorker Implementation for malicious purpose

In our proof-of-concept implementation for the malware persistence with ServiceWorker, we implemented a MapReduce system. In this malicious MapReduce system, all the current infected web browsers receive the map function and a chunk of the data via WebSocket. The map function executes as a ServiceWorker and operates over the data chunks sent by the malicious server. When the ServiceWorker finishes executing the map function, it returns the result to the malicious server via WebSocket. When the malicious server receives the results from the ServiceWorker, it performs the reduce phase and returns the final result to the malware author.

3.4.3 Attack Vectors

The ability to inject and execute arbitrary JavaScript code allows JSless to support a wide variety of malicious attacks. Here are the most common attacks that JSless could execute:

3.4.3.1 Data Stealing

On infection JSless can easily collect keystrokes, cookie and web storage data, as demonstrated in our PoC. Also, it could control multimedia devices and capture data from a connected mic or webcam using native browser WebRTC APIs.

3.4.3.2 DDoS

JSless malicious C&C server could orchestrate all the currently infected web browsers to connect to a specific URL or web server to perform a DDoS attack. In this case, JSless constructs a botnet of infected browsers to execute the DDoS attack.

3.4.3.3 Resource Consumption Attack

In this case, JSless could use the infected users' browser to run computationally intensive tasks such as cryptocurrency mining, password cracking, etc. The MapReduce system we implement as part of JSless is an example of managing and running computationally intensive tasks. Also, beside the above attacks which we have implemented in our JSless it is possible to perform other attacks like Click Fraud, RAT-in-the-Browser (RitB) Attacks, and many other web-based attacks.

3.5 Experiment & Evaluation

In order to assess the identified JavaScript/HTML5 vulnerabilities and threats, we developed JSless as a proof-of-concept fileless malware that is completely written in JavaScript. We used the second injection scenario to test our fileless malware implementation. For this purpose, we also implemented a web app that JSless will compromise to infect the web browser of any user using the web app. The web app is a shared chat board that allows users to register, post and receive messages to/from a shared chat board. The web app and the JSless C&C server are implemented in JavaScript using MEAN stack (MongoDB, ExpressJS, AngularJS, and Node.js). The source code for the fileless malware and the target web app is available on our GitHub/bitbucket repository for interested researchers and security analysts.

For the actual test, we deployed the target web app and the JSless C&C server on Amazon Web Services (AWS). We used two AWS instances with two different domains, one to host the target web app and the second to host JSLess C&C server. We mainly tested two attack vectors, the data stealing attack and the resource consumption attack.

3.5.1 JS Malware Detection Tools

To our surprise, few anti-malware systems try to detect JavaScript malware. We identified seven tools that we considered promising based on the techniques and the technology they use for detection. Most of the tools apply both static and dynamic analysis. Some of those tools are commercial, but they provide a free trial period that includes all the commercial feature for a limited time. Table 3.5.1 shows the list of tools we used in our study.

Tool Name	Detection Technique	License	Website	Detect JSLess
ReScan.pro	static & dynamic	commercial	https://rescan.pro/	NO
VirusTotal	static & dynamic	free & commerical	https://www.virustotal.com/	NO
SUCURI	static	commercial	https://sucuri.net/	NO
SiteGuarding	static	commercial	https://www.siteguarding.com/	NO
Web Inspector	static & dynamic	free	https://app.webinspector.com/	NO
Quttera	static & dynamic	free & commercial	https://quttera.com/	NO
AI-Bolit	static & dynamic	free & commercial	https://revisium.com/aibo/	NO

Table 3.5.1: JavaScript and Web App Malware Detection Tools

None of the tools were able to detect JSless malicious behaviors. To confirm our results we invited different teams from anti-malware service providers to inspect our compromised web app. Only Fortiguard Labs (https://fortiguard.com/) confirmed the malicious behaviors of JSless through manual analysis and full access to the obfuscated source code of JSless since the automated tools raised a suspicious flag.

3.5.2 Detection & Mitigation

By reviewing the results from the detection tools and how those tools work, it is obvious that detecting JSLess is very difficult. The use of WebSocket to inject and run obfuscated malicious code makes it almost impossible for any static analysis tool to detect JSLess, since the malicious payload does not exist at the time of static analysis. The use of benign JavaScript/HTML5 APIs and features, in addition to the dynamic injection behaviors, also make it very difficult for the current dynamic analysis tools

to detect JSLess. Blocking or preventing new JavaScript/HTML5 APIs is not the solution and it is not an option. In our opinion, a dynamic analysis technique that implements continuous monitoring and is context-aware is the only approach that we think could detect or mitigate fileless malware similar to JSLess.

3.5.3 Detection Tool Analysis Results

ReScan.Pro

ReScan.Pro is a cloud-based web application scanner which takes the URL of a website and generates a report after scanning the website for web-based malware and other web security issues. It explores the website and checks for infections, suspicious content, obfuscated malware injections, hidden redirects and other web security threats present. Analysis by ReScan.Pro is based on three main features.

1. *Static Page Scanning:* A combination of generic signature detection techniques and heuristic detection. It uses signature and pattern-based analysis to identify malicious code snippets and malware injections. It also looks for malicious and blacklisted URLs in a proprietary database.

2. *Behavioral Analysis:* It imitates the website user's possible behavior to evaluate the intended action of implemented functionality.

3. *Dynamic Page Analysis:* performs dynamic web page loading analysis which includes deobfuscation techniques to decode the obfuscated JavaScript in order to identify runtime code injections and check for malware in external JavaScript files.

We ran the experiment with the ReScan.Pro to test if it will detect the malicious activities of JSless malware. It generated a well defined report after analyzing the website with its static and dynamic features. The produced result indicated the website is clean and no malicious activity has been found. ReScan.Pro could not detect our JavaScript fileless malware.

Web Inspector

This tool runs a website security scan and provides a report of a given website after it is provided with its URL. Its security scanner is bit different from others because it performs both malware and vulnerabilities scans together. This tool claims to provide five different detection techniques; Honeypot Engine, Antivirus Detection, BlackList Checking, SSL Checking, and Analyst Research.

In our experiment our JavaScript fileless malware was able to successfully deceive this malware detection tool as well. Web Inspector's report indicated that no malware was detected.

Sucuri

Sucuri is another tool that offers a website security evaluation with a free online scanner. This scanning tool searches for various indicators of compromise, which includes malware, drive-by downloads, defacement, hidden redirects, conditional malware, etc. Sucuri claim to uses static techniques with intelligent signatures which are based on code anomalies and heuristic detection to detect malicious behaviour. Server side monitoring is another service provided by them which can be hosted on the compromised server to look for backdoors, phishing attack vulnerabilities, and other security issues by scanning the files present on the server. Moreover, Sucuri also provides a scanning API as a paid feature.

Testing Sucuri online scanner with JSLess, we found that it failed to detect out fileless malware, indicating that there is "No Malware Found" as well as indicating a medium security risk. However, this is due to Insecure SSL certificates, not from the detection of our fileless malware.

Quttera

Quttera is yet another website scanner that attempts to identify malware and suspicious activities in web applications. Its malware detector contains non-signature based approaches which attempt to uncover traffic re-directs, generic malware, and

security weakness exploits. It also claims to provide real-time detection of shell-codes, obfuscated JavaScript, malicious iframes, traffic re-directs and other threats. Here too, the website scanner failed to detect our JavaScript fileless malware.

VirusTotal

VirusTotal is a popular free malware inspection tool which offers a number of services including websites scanning. They aggregate different tools which cover a wide variety of techniques, such as heuristic, signature based analysis, domain blacklisting services, and more. A detailed report is provided after completing the scan which not only indicates the malicious content present in a website but also exhibits the detection label by each engine.

We scanned our compromised web app with VirusTotal which used 66 different malware detection engines, and none of were able to detect that the web app is compromised, as shown in figure.

AI-BOLIT

AI-BOLIT is an antivirus/malware scanner for website browsing and hosting. It uses heuristic analysis and other "patented AI algorithms" to find malware. We used it to scan our JSLess malware scripts. However, it failed to detect JSLess and generated a false positive when it consider some of the core modules of NodeJS as malicious JavaScripts.

3.6 Conclusion & Future Work

In this chapter, we confirmed several threat-vectors that exist in new JavaScript and HTML5 features. We demonstrated how an attacker could abuse benign features and APIs in JavaScript and HTML5 to implement fileless malware with advanced evasion capabilities. We showed a practical implementation of a fileless JavaScript malware that to our knowledge is the first of its kind. The proof-of-concept implementation of the proposed JS fileless malware successfully bypasses several well-known

anti-malware systems that are designed to detect JavaScript and web malware. In addition, third-party malware analyst teams confirmed our finding and proved that the proposed malware bypasses automated malware detection systems. From this particular study, we conclude that the current static and dynamic analysis techniques are limited if not useless against fileless malware attacks. Moreover, fileless malware attacks are not limited to PowerShell and Windows environment. In our opinion, any computing environment that enables running and executing arbitrary JavaScript code is vulnerable to fileless attacks.

Our future work could be summarized in three different directions. First, we will continue extending the malicious behaviors of JSLess and investigate the possibility of more advanced attacks using other new benign features and APIs from JavaScript and HTML5. Second, we will design a new detection technique to detect advanced JS malware and mainly fileless JS malware like the proposed JSLess. We plan to implement dynamic analysis approaches that continually monitor and analyze JavaScript and Browser activities. Finally, our third research direction will focus on investigating the fileless malware threat in unconventional computing environments, such as the Internet of Things, in-memory computing environments (e.g., Redis, Hazelcast, Spark, etc.), and so on. We hope our research will help to raise awareness of the emerging unconventional malware threats.

References

[1] H. Adas, S. Shetty, and W. Tayib. "Scalable detection of web malware on smartphones". In: *2015 International Conference on Information and Communication Technology Research (ICTRC)*. May 2015, pp. 198–201. DOI: 10.1109/ICTRC.2015.7156456.

[2] G. Blanc et al. "Characterizing Obfuscated JavaScript Using Abstract Syntax Trees: Experimenting with Malicious Scripts". In: *2012 26th International Conference on Advanced Information Networking and Applications Workshops*. Mar. 2012, pp. 344–351. DOI: 10.1109/WAINA.2012.140.

[3] D. Cosovan, R. Benchea, and D. Gavrilut. "A Practical Guide for Detecting the Java Script-Based Malware Using Hidden Markov Models and Linear Classifiers". In: *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. Sept. 2014, pp. 236–243. DOI: 10.1109/SYNASC.2014.39.

[4] Y. Fang et al. "Research on Malicious JavaScript Detection Technology Based on LSTM". In: *IEEE Access* PP (Oct. 2018), pp. 1–1. DOI: 10.1109/ACCESS.2018.2874098.

[5] Global Research and Analysis Team, KASPERSKY Lab. *Fileless Attack Against Enterprise Network*. White Paper. KASPERSKY Lab, 2017.

[6] K. R. Kishore et al. "Browser JS Guard: Detects and defends against Malicious JavaScript injection based drive by download attacks". In: *The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014)*. Feb. 2014, pp. 92–100. DOI: 10.1109/ICADIWT.2014.6814705.

[7] A. Magnusardottir. *Fileless Ransomware: How It Works & How To Stop it?* White Paper. Cyren, June 2018.

[8] D. Maiorca et al. "Detection of Malicious Scripting Code Through Discriminant and Adversary-Aware API Analysis". In: *Proceedings of the First Italian Conference on Cybersecurity (ITASEC17), Venice, Italy, January 17-20, 2017.* Ed. by Alessandro Armando, Roberto Baldoni, and Riccardo Focardi. Vol. 1816. CEUR Workshop Proceedings. CEUR-WS.org, 2017, pp. 96–105. URL: http://ceur-ws.org/Vol-1816/paper-10.pdf.

[9] J. Mao et al. "Detecting Malicious Behaviors in JavaScript Applications". In: *IEEE Access* 6 (2018), pp. 12284–12294. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2795383.

[10] McAfee. *Fileless Malware Execution with PowerShell Is Easier than You May Realize*. McAfee, Mar. 2017. URL: https://www.mcafee.com/enterprise/en-us/assets/solution-briefs/sb-fileless-malware-execution.pdf.

[11] S. Ndichu et al. "A Machine Learning Approach to Malicious JavaScript Detection using Fixed Length Vector Representation". In: *2018 International Joint Conference on Neural Networks (IJCNN)*. July 2018, pp. 1–8. DOI: 10.1109/IJCNN.2018.8489414.

[12] S. Oh et al. "Malicious Script Blocking Detection Technology Using a Local Proxy". In: *2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS)*. July 2016, pp. 495–498.

[13] V. Sachin and N. N. Chiplunkar. "SurfGuard JavaScript instrumentation-based defense against Drive-by downloads". In: *2012 International Conference on Recent Advances in Computing and Software Systems*. Apr. 2012, pp. 267–272.

[14] B. Sayed, I. Traoré, and A. Abdelhalim. "Detection and mitigation of malicious JavaScript using information flow control". In: *2014 Twelfth Annual International Conference on Privacy, Security and Trust*. July 2014, pp. 264–273. DOI: 10.1109/PST.2014.6890948.

[15] P. Seshagiri, A. Vazhayil, and P. Sriram. "AMA: Static Code Analysis of Web Page for the Detection of Malicious Scripts". In: *Procedia Computer Science* 93 (2016). Proceedings of the 6th International Conference on Advances in Computing and Communications, pp. 768–773. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2016.07.291. URL: http://www.sciencedirect.com/science/article/pii/S187705091631537X.

[16] V. R. L. Shen, C. Wei, and T. Tong-Ying Juang. "Javascript Malware Detection Using A High-Level Fuzzy Petri Net". In: July 2018, pp. 511–514. DOI: 10.1109/ICMLC.2018.8527036.

[17] I. A. AL-Taharwa et al. "RedJsod: A Readable JavaScript Obfuscation Detector Using Semantic-based Analysis". In: *2012 IEEE 11th International Conference on Trust, Security and Privacy in Computing and Communications*. June 2012, pp. 1370–1375.

[18] *The 2017 State of Endpoint Security Risk*. White Paper. Ponemon Institute, 2017.

[19] TrendMicro. *Analyzing the Fileless, Code-injecting SOREBRECT Ransomware*. TrendMicro, June 2017. URL: https://blog.trendmicro.com/trendlabs-security-intelligence/analyzing-fileless-code-injecting-sorebrect-ransomware/.

[20] C. Wang and Y. Zhou. "A New Cross-Site Scripting Detection Mechanism Integrated with HTML5 and CORS Properties by Using Browser Extensions". In: *2016 International Computer Symposium (ICS)*. Dec. 2016, pp. 264–269. DOI: 10.1109/ICS.2016.0060.

[21] Y. Wang, W.g Cai, and P. Wei. "A deep learning approach for detecting malicious JavaScript code". In: *Security and Communication Networks* 9 (2016), pp. 1520–1534.

[22] W. Xu, F. Zhang, and S. Zhu. "The power of obfuscation techniques in malicious JavaScript code: A measurement study". In: *2012 7th International Conference on Malicious and Unwanted Software*. Oct. 2012, pp. 9–16.

[23] S. Yoon et al. "Automatic attack signature generation technology for malicious javascript". In: *Proceedings of 2014 International Conference on Modelling, Identification Control*. Dec. 2014, pp. 351–354. DOI: 10.1109/ICMIC.2014.7020779.

CHAPTER 4

Interpreting Machine Learning Malware Detectors Which Leverage N-gram Analysis

William Briguglio and Sherif Saad

In Proceedings of the 12th International Symposium on Foundations and Practice of Security

4.1 Introduction

Adopting sophisticated machine learning techniques for malware detection or other cyber attack detection and prevention systems in a production environment is a challenge. This is because most of the time it is not possible to understand how machine learning systems make their detection decisions. In the malware detection domain, machine learning models can be trained to distinguish between benign binaries and malware, or between different malware families. The advantage of using machine learning models is that they are less sensitive to minute changes in malware binaries and can therefore detect unseen samples so long as they are designed and trained to detect characteristics common across seen and unseen samples. Furthermore, their learnt relationships can be used to determine relevant features for a classification, limiting the amount of data malware analyst must sift through to determine the functionality of a malicious binary. However, there are several drawbacks that must be addressed before their full potential can be realized in the malware detection domain. Firstly, due to the quick evolving nature of malware, the models must be made

efficient to train and update frequently when new malware families are discovered. Secondly, it is possible to create specially crafted "adversarial samples" which take advantage of peculiarities in the models learnt relationships to bypass the detector with relatively inconsequential changes to the binary. Finally, given the high degree of risk involved with classification errors, the models must provide a reason for their decisions in order to improve performance and increase trust in the model and its predictions.

The process of providing reasons for a machine learning model's predictions is known as interpretation. Interpretation in this setting should provide several key benefits. Firstly, due to the high cost of classification error, a low false positive and false negative rate is a must, and therefore these systems must be robust. Further, robustness makes it more difficult for malware authors to create adversarial malware to bypass the detector. A model is said to be robust if small changes in input do not cause large changes in output such as a different classification. Second, the high risk necessitates a high degree of model confidence. Therefore, interpretation must provide evidence that the model has learnt something which can be corroborated with industry knowledge. This also goes hand in hand with the first requirement as an interpretation which can show a model is robust can improve model confidence as well. Additionally, the interpretation should aid malware analysts in down stream tasks such as determining the functionality of a malware binary.

Machine learning interpretation can be broadly separated into two categories. One is model agnostic techniques which are independent of the type of model which they are interpreting and rely solely on the input and output of the model. The other, which we will be using in this chapter, are model specific techniques, which use specific elements of the model such as learnt weights or decision rules in order to provide an interpretation of a prediction. Interpretations themselves can be divided into global and local interpretations. Global interpretations provide an interpretation that is applicable across the entire feature space. Meanwhile local interpretations apply to only a single example or a small subset of the feature space. Some interpretation techniques provide only one type of interpretation while others provide both.

In this chapter we explore the interpretability of machine learning based malware classifiers in relation to the goals of model robustness, confidence in model predictions, and aiding the process of determining the functionality of a malware sample. We train a logistic regression model, random forest, and a neural network on a Microsoft data set containing the hexadecimal representations of malware binaries belonging to several different malware families. We then apply model specific interpretation techniques to provide both a global and local interpretation of each of the models. The objective of this chapter is to demonstrate interpretability techniques in practice on machine learning based malware detectors. We also try to evaluate the effectiveness of existing interpretability techniques in the malware analysis domain in terms of their usefulness to malware analysts in a practical setting. To the best of our knowledge, this is the only work which explores the application of machine learning interpretability techniques in the malware analysis domain.

4.2 Literature Review

In the last decade, with the increasingly massive data sets machine learning algorithms are being used on, and the growing complexity of the algorithms, the prediction process of these algorithms has become so non-intuitive that traditional analysis techniques no longer suffice. Analysis being necessary for a number of practical and legal concerns has caused research to now shift towards machine learning interpretability.

Christoph Molnar [11] put together a summary of machine learning interpretation methods in which he outlines a basic approach for the interpretation of Linear Regression models (of course the same approach can be applied to linear SVM's, Shirataki et al. [18]) where a feature's contribution to a prediction is the product of its value and weight. For logistic regression he shows that when the j^{th} feature value is incremented by 1, then the quotient of the predicted odds of the sample belonging to the positive class after the increase over the predicted odds of the sample belonging to the positive class before the increase is equal to e^{β_j}, where β_j is the weight of feature j. Alternatively, this means that a unit increase in feature j results in the

predicted odds increasing by $((e^{\beta_j} - 1) * 100)\%$. He goes on to discuss the seemingly trivial interpretation of decision trees as the conjunction of the conditions described in the nodes along a predictions path to a leaf node. Similarly, for rule list models, an "explanation" is simply restating the rule or combination of rules which lead to a decision.

However, the evaluation of a model's complexity is closely tied with its explanation's comprehensibility, especially for rule set models, linear models, and tree models. Given the following complexity definitions, the explanation approaches discussed above could be too complex for highly dimensional datasets. Marco Ribeiro et al. [15] define the complexity of a linear model as the number of non-zero weights and the complexity of a decision tree as the depth of the tree. Meanwhile, Otero and Freitas [12] defined the complexity of a list of rules as the average number of conditions evaluated to classify a set of test data. They referred to this as the "prediction-explanation size".

There has also been work done on the interpretability of neural networks(NNs) such as the Layer-wise Relevance Propagation introduced in [3] as a set of constraints. The constraints ensure that the total relevance is preserved from one layer to another as well as that the relevance of each node is equal to the sum of relevance contributions from its input nodes which in turn is equal to the sum of relevance contributions to its output nodes. Any decomposition function following these constraints is considered a type of Layer-wise Relevance Propagation. In [19], Shrikumar et al. propose DeepLIFT which attributes to each node a contribution to the difference in prediction from a reference prediction by back propagating the difference in predication scaled by the difference in intermediate and initial inputs.

Moving on to model agnostic methods, Friedman in [6] used Partial Dependence Plots (PDP) to show the marginal effect a feature has in a predictive model. Similarly, Goldstein et al. [8] used Individual Conditional Expectation (ICE) plots to show a curve for each sample in the data set where one or two features are free variables while the rest of the features remain fixed. Since ICE plots and PDPs do not work well with strongly correlated features, Deniel W. Apley et al. [2] proposed Accumulated

Local Effects plots to display the average local effect a feature has on predictions.

The H-statistic was used by Friedman and Popescu in [7] (equations 44-46) to provide a statistical estimate of the interaction strength between features by measuring the fraction of variance not captured by the effects of single variables. Feature Importance was measured by Breiman [4] as the increase in model error after a feature's values are permuted (a.k.a. permutation importance).

Marco Ribeiro et al. in [15] defined a version of the surrogate method which can explain individual predictions using an approach called Local Interpretable Model-agnostic Explanations (LIME) which trains an interpretable classifier by heavily weighing samples nearer to a sample of interest. Tomi Peltola [13] extended this work with KL-LIME, which generated local interpretable probabilistic models for Bayesian predictive models (although the method can also be applied to non-Bayesian probabilistic models) by minimizing the Kullback-Leibler divergence of the predictive model and the interpretable model. This has the added benefit of providing explanations that account for model uncertainty. Strumbelj et al. [20] detailed how to describe the contributions made by each feature to a prediction for a specific instance using Shapely Values, a concept adopted from coalitional game theory.

Finally, there are Example-Based methods such as the method put forward by Wachter et al. in [21] which produce interpretations by finding counter-factual examples which are samples with a significant difference in prediction, whose features are relatively similar to the sample of interest, by minimizing a loss function. The found sample is then used to explain what small changes would cause the original prediction to change meaningfully. There is also the MMD-critic algorithm by Kim et al. [9] which finds Prototypes (well represented examples) and Criticisms (poorly represented examples) in the dataset. To find examples in the training data which have a strong effect on a trained linear regression model (i.e. influential instances) Cook [5] proposed Cook's distance, a measure of the difference in predictions made by a linear regression model (however the measure can be generalized to any model) trained with and without an instance of interest. Koh and Liang [10] put forward a method for estimating the influence of a specific instance without retraining the

model as long as the model has a twice differentiable loss function.

4.3 Method

Training and classification were done on a data set of 10,896 malware files belonging to 9 different malware families.[1] The data set is discussed in [16]. Each sample consists of the hexadecimal representation of the malware's binary content. The class details are summed up in table 4.3.1.

Table 4.3.1: Class distribution in Data Set

Class No.	Family	Sample Count	Type
1	Ramnit	1541	Worm
2	Lollipop	2478	Adware
3	Kelihos_ver3	2942	Backdoor
4	Vundo	475	Trojan
5	Simda	42	Backdoor
6	Tracur	751	TrojanDownloader
7	Kelihos_ver1	398	Backdoor
8	Obfuscator.ACY	1228	obfuscated malware
9	Gatak	1013	Backdoor

Based on other work on the the same data set, we decided to use n-grams as features. N-grams are sequences of words of length n which occur in a body of text. However, in our case the n-grams are sequences of bytes of length n which occur in a binary. The length of n-gram we settled on was 6 because they were shown to preform well in [14], however our approach can work with n-grams of arbitrary length. We extracted the 6-gram features from the hex representations of the malware files by

[1]The data set was downloaded from `https://www.kaggle.com/c/malware-classification/data`

obtaining the entire list of 6-grams present in the data set, and the number of files each 6-gram appeared in. This resulted in over 2,536,629,413 candidate features. Next, any 6-gram which did not appear in at least 100 files was removed from consideration, bringing the feature set size down to 817,785. This was done because [14] also showed that selection by frequency is an effective way to reduce the initial feature set size and a computationally cheap approach was needed considering the number of features.

Next, feature vectors were created for each of the malware samples so that a more sophisticated feature selection method can be preformed. This was done by searching for the selected 6-gram feature in a binary and setting the corresponding value in that binary's feature vector to 1 if the binary did contain the 6-gram, and 0 otherwise. To select the features for the logistic regression model, Chi^2 was used because it can detect if a categorical feature is independent of a predicted categorical variable (in this case our class) and is therefore irrelevant to our classifier. For the neural network and random forest, Mutual Information (MI) was used because it can detect the more complex dependencies between a feature and a sample's classification which can be taken advantage of by a neural network or random forest. Since the feature set was still very large, the Chi^2 and MI scores had to be calculated in batches. This was done by splitting the data set into 20 batches, each with the same distribution of classes, and averaging out the resulting scores for each feature. Next, the features with Chi^2 scores above 330 or MI scores above 0.415 were selected. This brought the feature set size down to 8867 in the case of the logistic regression model and to 9980 in the case of the neural network and random forest. The feature set size was determined based off other work using n-grams to classify the same data set. We did not attempt to find an optimal feature set size as our primary focus was model interpretation.

Next, the models were trained on their respective feature sets. To find the best parameters for the logistic regression model and train the model, grid search with 5-fold cross validation was used, yielding $C = 10$ and tolerance $= 0.0001$. The value of C inversely determines the strength of regularization, that is, smaller values of C cause more feature weights in the classifier to be set to 0, a value of 0 corresponds to no regularization, and values above 0 encourage the classifier to use more fea-

tures. Tolerance determines the minimum change in error, from one iteration of the optimization algorithm to the next, that causes the algorithm to terminate training. Similarly for random forest, finding the best parameters and training was done with grid search with 5-fold cross validation as well. The number of trees found to preform best was 300 and the and the minimum samples per leaf found to preform best was 0.01% of the total number of samples. The grid search with cross validation, logistic regression model, and the random forest model were implemented using the scikit python library.

For the neural network the data was split into a training and a test set each with the same class distribution. This was done because the extra parameters in a neural network require a larger data set to learn more abstract patterns and splitting it up into many folds might have stifled this process. The neural network consisted of an input layer with one neuron per feature, an output layer with one neuron per class using the sigmoid activation function, and a hidden layer consisting of 40 neurons using the tanh activation function. 40 neurons was chosen because that number was found to preform the best after testing with various other configurations. There were also no bias units to aid in interpretation. The neural network was implemented using the Keras python library.

After training and testing the three models, the logistic regression model was interpreted by examining the weights used by the classifier. The random forest was interpreted by examining the feature importance as well as using the treeInterpreter python library [17] to obtain feature contributions to a particular prediction. In the case of the Neural network, the iNNvestigate python library by [1] was used to preform LRP to get the relevances of each node in the model for interpretation. The balanced accuracy on the left out fold was 96.19% for the logistic regression model and 96.97% for the random forest. The balanced accuracy on the test set was 94.22% for the neural network. A discussion of the model interpretations follows in the next section.

4.4 Interpretation

4.4.1 Logistic Regression Model Interpretation

The logistic regression model uses a one-vs-rest classification scheme whereby for each class, a constituent model is trained to classify a sample as either that class, or not that class, and therefore we are actually dealing with nine separate logistic regression models each making binary classifications. For this reason we cannot preform the typical global interpretation of the overall multi-class model by examining the weights since the weights should be different for each of the binary models. However, we can gain insight of the importance of each feature by averaging these weights across the 9 constituent binary models. For this we take the average of the absolute values of the weights. This is because if a feature contributes positively for one constituent binary classifier and negatively for another, then the weights would cancel each other out during averaging which would falsely give the impression that the feature was not important in the overall multi-class model. Table 4.4.1 shows the largest 15 averages of the absolute feature weights.

Looking at the table 4.4.1, we can see that three 6-grams are relatively heavily weighted, 00E404000000, 0083C4088B4D, and C78530FDFFFF. Recall from section 4.2 that for logistic regression, when the j^{th} feature value is incremented by a value of 1, then the predicted odds increase by $((e^{\beta_j} - 1) * 100)\%$, where β_j is the learnt weight of the j^{th} feature. In our case we are using binary feature values where a 1 indicates the presence of a 6-gram and 0 indicates its absence, so we interpret the weights as follows. When the 6-gram corresponding to the j^{th} feature is present, the predicted odds increase by $((e^{\beta_j} - 1) * 100)\%$. One may be tempted to apply this to the weights in table 4.4.1, but these are averaged *absolute* weights across all 9 constituent binary classifiers. Further, negative weights do not cause a decrease in the predicted odds that is proportional to a positive weight with the same absolute value due to the shape of the function $f(x) = e^x - 1$. Therefore, it would be inaccurate to say the average absolute effect of some 6-gram corresponds to a $(e^{avg_j} - 1)\%$ change in the predicted odds, where avg_j is the average absolute weight of feature j. Thus a global

Table 4.4.1: Max 15 Absolute Weights of the Logistic Regression Model Averaged Across All 9 Binary Sub-classifiers

Avg. Abs. Weight	Feature
1.3151053659364556	0000000066C7
1.3480135328294032	008B4C240C89
1.4629237676020752	8BEC83EC10C7
1.4846778818947817	00000000EB07
1.5276044995023308	B80000000050
1.540535475655897	500147657453
1.5605614219830626	006800004000
1.6494330450079937	89852CFDFFFF
1.685741868293823	0033C58945FC
1.7235671007282005	8B91C8000000
1.781357432072784	034C6F61644C
1.8232074423648363	8BEC6A006A00
2.071327588344743	00E404000000
2.15007904223129	0083C4088B4D
2.1561672884172056	C78530FDFFFF

interpretation of a multi-class one-vs-rest logistic regression model using n-grams in confined to vague statements about which n-grams are important based solely off their average absolute weights, which is not very useful in a practical setting.

Next we will examine the max weights for a constituent binary model. This will allow us to make conclusions on what features the model uses to detect a specific class of malware in the data set. Furthermore, we will be able to determine exactly the change in predicted odds that the presence of an n-gram causes. For the sake of brevity, we will examine just the binary model for class 3, corresponding to the Kelihos_ver3 family of malware, as all three models performed well for this class but the same process can be followed for the other constituent binary models corresponding to other classes. Table 4.4.2 shows the max 15 weights of the classifier for class 3.

Table 4.4.2: Max 15 Weights for Kelihos_ver3 Binary Sub-classifier

Weight	Feature
0.6438606978376447	000607476574
0.6438606978376447	000C07476574
0.6438606978376447	060747657444
0.6438606978376447	074765744443
0.6438606978376447	0C0747657444
0.6438606978376447	930644697370
1.3719246726968015	00000083FEFF
1.5114878196031336	E8000000895D
2.1067800174989904	0F85CC010000
2.3123117293223405	0A0100008B45
2.9041700918303084	000F859D0000
3.174276823535364	000F84700100
3.5334477027408613	0083C4088B4D
3.7941081330633857	034C6F61644C
4.391600387291376	00008B5DE43B

In table 4.4.2 we can see three 6-grams have relatively large weights. This means these n-grams are most strongly associated with class 3 and in this case, since we are looking at only the weights for a single binary classifier, we can use our interpretation from above. That is, when the 6-gram corresponding to the j^{th} feature is present, the predicted odds increase by $((e^{\beta_j} - 1) * 100)\%$. For example we can say the presence of 00008B5DE43B, increases the predicted odds of a sample belonging to class 3 by $((e^{4.3916} - 1) * 100)\% = 7977\%$. At first glance this number may seem excessive but in order to make good sense of it we must also determine what the predicted odds of a sample belonging to class 3 are when this 6-grams are not present, using a reference sample. For this we use a zero-vector corresponding to a sample where none of the 6-grams used as features are present. Since the dot product of a zero vector and the weight vector is zero, we only need to take the sigmoid of the intercept of the binary model for class 3 to determine the predicted probability of the reference vector belonging to class 3. The intercept is -4.2843, thus the predicted probability of the reference sample belonging to class 3 is $\text{sigmoid}(-4.28426) = 0.01360$. Next we must convert this to odds with $0.01360/(1 - 0.01360) = 0.01378$. This means a sample with no feature 6-grams present except 00008B5DE43B increases the odds from 0.01378 by 7977% to $0.01378 + (0.01378 * 79.77) = 1.11301$ predicted odds, or a 0.5267 predicted probability, of belonging to class 3. Thus we see that because of the intercept, the large weight of this feature does not necessarily guarantee a classification into class 3.

We can get a better idea of the robustness of the model by checking the number of 6-grams which play a significant role in the classification of a sample into class 3. This is because robustness is a measure of how tolerant a model is to small changes in input. Therefore, if the number of 6-grams which play a significant role is large, then a large number of changes in input will be required for a change in classification, thus giving us confidence in the model's robustness. However, if the number of significant features is low then only a small number of changes in input will be required for a change in classification, changes that may be easy and inconsequential for malware authors to make. Thus the robustness of the model would be called into question. In our case,

20 features have weights greater than or equal to about 0.59. 6-grams with weights above this number increase the predicted odds by $((e^{0.59} - 1) * 100)\% \approx 80\%$. Since the predicted odds of the reference example belonging to class 3 is 0.01378, this means about 11 such features can cause a sample to be classified as class 3 with about 90% predicted probability. This may indicate that the model is putting too much emphasis on just a few highly weighted 6-grams. To test this we can reclassify samples belong to class 3 with the highest weighted 6-grams set to 0. In our case, we set the nine highest weighted features to 0 for all samples. This required 22863 changes to the feature array, and the result was only 24 more misclassifications, 17 of which belonged to class 3, which has 2942 samples. Here, we encounter a specification issue. Currently, there is no formally defined metric to measure robustness quantitatively and once there is, a threshold for acceptable robustness will be application specific. We leave a definition of a robustness metric to future work, however, given that robustness is defined in terms of a model's tolerance to changes in input, and that tolerance to changes of insignificant features is irrelevant, we can be confident that this approach can give us an idea of our model's robustness. The models robustness becomes more clear when compared with other models. For example, if setting the same number of features to 0 in another model resulted in more or less misclassification, then we can say that model is less or more robust respectively than our logistic regression model Therefore, we can confidently say our approach gave an idea of model robustness for class 3. One can increase the model's robustness by further training the classifier with samples which have the highly weighted 6-grams removed. This would force the classifier to learn a more diverse set of features which correspond to class 3, meaning that more changes would be required to change a prediction to or from class 3. Thus by observing the important features, we can improve the models robustness to small changes in the input. A similar strategy can be followed for the most negatively weighted features. If there are features with too large negative weights, then a detector can be fooled by intentionally adding these 6-grams. Further training the classifier by adding the large negative weighted 6-grams to samples labeled class 3 will force the classifier to learn not to negate positively weighted features with one or a small set of 6-grams.

Therefore we can conclude that examining the weights in the manner we have done here can be useful for debugging logistic regression models leveraging n-grams. This interpretation is still global in that it encompasses the entire feature space, however, it must be repeated for each class. On the upside though, the global interpretation doubles as a local interpretation as the relationship between the presence of an n-gram and the change in the predicted odds holds across the entire data set for each sample.

Furthermore, this method for finding important n-grams features can be helpful in a practical setting as it can be used to aid malware analysts in down stream tasks. A malware binary's functionality can be more easily determined by implementing a method which automatically disassembles binaries and highlights the code which corresponds to the most heavily weighted n-grams that are present in the binary. This approach can also improve confidence in the model if the highlighted code's functionality is corroborated with industry knowledge. Both these advantages require another interpretation step of mapping feature values from the feature space to the domain space (i.e. mapping n-grams to the corresponding code) which is not the focus of this chapter. The downside to this interpretation approach is that it is specific to logistic regression models only, and unlike models such as neural networks or decisions tress, logistic regression models are not easily capable of learning more complex relationships between features and target values.

4.4.2 Random Forest Interpretation

In the case of the random forest, interpretation is more difficult. It is easy in a more general sense, in that we can get the feature importance scores, shown below in table 4.4.3, and use these to determine what features are generally most important, but getting a more fine grained interpretation is a challenge as the random forest is an ensemble of often hundreds of different decision trees.

Table 4.4.3 gives us a great idea of the model robustness. Since the total feature importance is always equal to 1, we can be sure that the model isn't relying on just a small number of features to make predictions because the 15 most important features

Table 4.4.3: Max Feature 15 Importances for Random Forest

Feature Importance	Feature
0.006877917709695	726573730000
0.007047751117095	7450726F6341
0.00723117607771	647265737300
0.007262894349522	558BEC83EC08
0.007377076296786	0064A1000000
0.007401045194749	727475616C41
0.007815881804511	A10000000050
0.008221953575956	75616C416C6C
0.008652467124996	634164647265
0.008657476622364	8A040388840D
0.008840768087294	69727475616C
0.008879491127129	89F5034C2404
0.00898170788833	7475616C416C
0.008987620418762	008A840D2F06
0.009011931204589	060000E2EFB9

only accounts for 0.9% of the total importance. Additionally, the feature importance steadily declines without one feature or a small group of features overshadowing the rest. Unfortunately, general statements about robustness which do not provide much utility to the malware analyst in a practical setting are the most we can say with a global interpretation. However considering a single example can give us more information, albeit only locally.

Interpretation of a Single Sample with Random Forest

With random forest, a local interpretation of a single example is difficult as a classification decision is the result of a vote amongst many different decision trees. However, here we find the tree with the highest predicted probability that a specific example belongs to its actual class. Then we use the tree interpreter library [17] to break down the contributions of each 6-gram feature. In our case we followed this procedure for sample 4WM7aZDLCmlosUBiqKOx and found that the 6-gram 002500000031 and the bias contributed 97.3% of the total feature importance. One may be tempted to think this means the model is relying on only a single feature however this is just one tree out of many which have heavily varying structures. Thus, changing this feature may not cause many of the other tree's predictions to change, such is the advantage of using random forests over single decisions trees. The significance of the resulting feature contribution is two fold. Firstly, the model designer can find the code corresponding to 002500000031 in the assembly code and determine weather the functionality of the code corroborates industry knowledge. If it does, then this can be used with other examples to improve model confidence. Secondly, by finding 6-grams in the constituent decision trees of the random forest model which are significant to a prediction, a process can be automated to disassemble the input file and highlight the code that corresponds to these significant 6-grams, aiding in malware analysis. The downside to this approach is that the random forest is made up of many different decision trees, many of which should all be predicting the correct class, so an automated process which collects significant 6-grams from these constituent trees and highlights the corresponding code may provide an overwhelming number of results.

This is because well over a hundred trees will be contributing at least a few 6-grams, meaning that potentially 100's of snippets of code will be highlighted to the analyst. Once again we are faced with the problem of mapping the feature values to the domain, however this should not be too tall a task and we leave this challenge to future work.

4.4.3 Neural Network Model Interpretation

For our global interpretation of the Neural Network model, we used LRP to determine the most relevant input nodes for classification. LRP was preformed in this experiment using iNNvestigate python library by [1]. First we found the relevances of the input nodes for each sample and then we averaged the absolute values of these relevances for the entire data set. This was done because one input node may be positively contributing to one output nodes prediction while negatively contributing to another, causing the input nodes relevances to cancel out during averaging and giving false impressions about the feature set. Table 4.4.4 shows the largest 15 averages of the absolute relevances.

In Table 4.4.4 we can see two values had significantly higher relevances than the rest, 000000000400 and 0000000000FF, and are therefore important for the models classification. Additionally, we can see many of the features which appear here are also in the top 15 most important 6-grams for the random forest. This result partially validates our technique for finding important 6-gram features in a neural network which to the best of our knowledge is a novel use of LRP in this domain. This gives us a general idea of the importance of features used by the model but, just like in the case of the other two models, we are still confined to vague general statements about a feature's importance. However, this time it is due to the complexity of the model.

Next we will examine the max relevances for a particular class. In this case we average the relevances for each node across all samples which were correctly classified as class 3. Table 4.4.5 shows the max 15 average relevances for class 3.

In Table 4.4.5 we can see five of the features which appear here are also in the top 15 highest weighted 6-grams for the binary logistic regression classifier for class

Table 4.4.4: Max Average Absolute Relevances

Avg. Abs. Relevance	Feature
0.4204155570273673	24000000008B
0.438576163384531	**75616C416C6C**
0.4604056179848827	000400000000
0.6358686047042836	00FFFFFFFFF
0.6414918343055965	**008A840D2F06**
0.6961477693970937	**060000E2EFB9**
0.7207968499760279	**8A040388840D**
0.7391062783969391	000001000000
0.7655264716760353	040000000000
0.7695977668414099	**89F5034C2404**
0.8623695409436033	416C6C6F6300
0.8762457266039623	6C6C6F630000
0.8811945910382549	**69727475616C**
1.1011308772023591	000000000400
1.129173981900078	0000000000FF

Bolded 6-grams also present in Table 4.4.3

Table 4.4.5: Max Avg Relevances for Class 3

Avg Relevance	Feature
0.07849652902147494	**060747657444**
0.0858714786617495	8B0000006700
0.08840799934653523	07497357696E
0.09155762345728868	**0C0747657444**
0.09213969967088875	F10448656170
0.09360746295067239	00F0F0280000
0.09471450612061977	00F104486561
0.10572475395119978	C3008BFF558B
0.10603324133390207	009306446973
0.11341626335133194	**000C07476574**
0.11451772113662628	C38BFF558BEC
0.12097247805393918	**930644697370**
0.14448647700726405	04546C734765
0.1895982317973578	064469737061
0.24520372442763907	**034C6F61644C**

Bolded 6-grams also present in Table 4.4.2

3. This result also partially validates our technique for finding important 6-gram features in a neural network for a single class. In this case we are still confined to general statements about a features importance for a specific class. However, we can get an idea of the model's robustness by setting the features with the highest average relevances for class 3 to 0 for all correctly classified samples in class 3. If the model relies heavily on only the presence of these 6-grams, then the class accuracy will drop drastically, however if we have a similar class accuracy as before, then it is unlikely that the features with a lesser average relevance would have a larger effect on the class accuracy and therefore we can somewhat confidently say the model is robust for this class. In our experiment the top 4 highest average relevance features were all set to 0 and it resulted in no further misclassifications. Therefore we can say our model is somewhat robust for class 3. This result is somewhat helpful in a practical setting as a malware analyst can use this technique to ensure the robustness of their model, but not much else.

Interpretation of a Single Sample with Neural Network

Next we'll further explore the neural network's predictions for samples belonging to class 3 by taking the test sample with the highest predicted probability of belonging to class 3, sample 4WM7aZDLCmlosUBiqKOx, and examining relevances for this sample in order to provide a local interpretation. In doing so we can see what the internal nodes are learning. First we determine the internal node relevances for this sample. The library used for this experiment did not have a built in method to determine the relevances of internal layer nodes so we created a second neural network that was a duplicate of the last two layers of the original neural network. We then obtained the value of the second layer nodes before the activation function is applied when classifying this sample. That is, if W^1 is the weight matrix for the connections between layer 1 and layer 2, and X^1 is the outputs of layer 1, then we obtained $X^1 \cdot W^1$. We then inputted $X^1 \cdot W^1$ into our second neural network and preformed LRP to get the relevances of the first layer of our second neural network which are equivalent to the relevances of the hidden layer in our original neural network. The

most relevant node by a substantial margin was the 40^{th} node in the hidden layer with a relevance of 0.61 and an activation of -0.99996614. Since this node is in layer 2 we will denote it with n^2_{40}. Next we created a third neural network that had two layers. The first was identical to layer 1 of our original neural network, the second was just the single node, n^2_{40}, from the original neural network, and the weight matrix for the connections from layer 1 to layer 2 of this new network is $W^1_{(40)}$ were $W^1_{(i)}$ is the 9980-dimensional weight vector for connections from layer 1 to the i^{th} node in layer 2 of the original neural network. In this way we were able to obtain the relevances of the input layer to only the activation of n^2_{40} in the hidden layer. Table 4.4.6 shows the max 10 node relevances for the activation of n^2_{40} in the hidden layer.

Table 4.4.6: Layer 1 Nodes relevance to n^2_{40}

Activation	Relevance	Feature
1.0	0.025721772	007300000061
1.0	0.027428055	230000001900
1.0	0.02751717	2F0000002300
1.0	0.029254071	270000003300
1.0	0.030343212	00870000009D
1.0	0.03163522	002F00000025
1.0	0.031697582	040000C00000
1.0	0.03176714	002300000019
1.0	0.032007236	00C0000000D0
1.0	0.034308888	007701476574

In table 4.4.6 we can see that many of the 6-grams have similar relevance's which slowly decrease. This corroborates our results when examining class 3 as a whole since the similar relevances across many input nodes indicates that many features are responsible for a classification which is to be expected when a model is robust

to changes in the input data. One can automate the process of preforming LRP on specific examples to find relevant input nodes, both for the entire model and for a specific internal node possibly showing what the internal node is learning. From there highlighting the disassembled code which corresponds to the most relevant nodes can help malware analyst either determine the functionality of the file or show that the model has learnt something which corresponds to industry knowledge, thus improving confidence in the model.

4.5 Conclusion

In this chapter we demonstrated techniques for the interpretation of malware detectors which leverage n-grams as features. We've shown that it is possible to interpret a neural network, a logistic regression model, and a random forest, with the objectives of debugging and creating robust models, improving model confidence, and aiding malware analysts in downstream tasks. For the logistic regression model, examining the weights was all that was needed to meet these goals. However, although straight forward to interpret, the model was less expressive then the other two considered. The random forest required slightly more work for analysis but it was also possible to get a meaningful local interpretation that helped with the above stated goals. The downside here was that the random forest interpretation must consider many of the constituent trees to be thorough, which can be time consuming and provide too verbose results. The neural network interpretation was much more intensive but by using layer-wise relevance propagation it was possible to determine the relevance or significance of different n-grams across the data set, across a specific class, and for a single example or for a single node. Thus, we were able to provide a global and local interpretation which was somewhat useful in a practical setting since by using these relevances it was then possible to get an idea of the robustness of the model and build confidence or aid in downstream analysis of samples.

Over all it was possible to satisfy our interpretation objectives for each model but the ubiquitous trade off between the interpretability and the expressivity of the

model was still present. Additionally, n-grams in and of themselves seem slightly problematic as it is not easy to determine what a n-gram corresponds to on its own, without considering a single example for context. So providing a global interpretation of a n-gram in order to show what the model has learnt is difficult. To this end it would be advantageous to include human readable features as well or other features which can be easily interpreted in a manner that doesn't require examining a specific real example.

For future work the interpretation of other models using other feature sets is a must. Additionally, a metric to quantify the robustness of a malware detector is needed for more direct comparison.

References

[1] M. Alber et al. *iNNvestigate neural networks!* 2018. arXiv: 1808.04260 [cs.LG].

[2] A. W. Apley and J. Zhu. *Visualizing the Effects of Predictor Variables in Black Box Supervised Learning Models*. 2016. arXiv: 1612.08468 [stat.ME].

[3] S. Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PloS one*. 2015.

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

[5] R. Dennis Cook. "Detection of Influential Observation in Linear Regression". In: *Technometrics* 19.1 (1977), pp. 15–18. DOI: 10.1080/00401706.1977.10489493. eprint: https://doi.org/10.1080/00401706.1977.10489493. URL: https://doi.org/10.1080/00401706.1977.10489493.

[6] J. H. Friedman. "Greedy function approximation: A gradient boosting machine." In: *Ann. Statist.* 29.5 (Oct. 2001), pp. 1189–1232. DOI: 10.1214/aos/1013203451. URL: https://doi.org/10.1214/aos/1013203451.

[7] J. H. Friedman and B. E. Popescu. "Predictive learning via rule ensembles". In: *Ann. Appl. Stat.* 2.3 (Sept. 2008), pp. 916–954. DOI: 10.1214/07-AOAS148. URL: https://doi.org/10.1214/07-AOAS148.

[8] A. Goldstein et al. *Peeking Inside the Black Box: Visualizing Statistical Learning with Plots of Individual Conditional Expectation*. 2013. arXiv: 1309.6392 [stat.AP].

[9] B. Kim, R. Khanna, and O. O. Koyejo. "Examples are not enough, learn to criticize! Criticism for Interpretability". In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee et al. Curran Associates, Inc., 2016, pp. 2280–2288. URL: http://papers.nips.cc/paper/6300-examples-are-not-enough-learn-to-criticize-criticism-for-interpretability.pdf.

[10] P. W. Koh and P. Liang. "Understanding Black-box Predictions via Influence Functions". In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70*. ICML'17. Sydney, NSW, Australia: JMLR.org, 2017, pp. 1885–1894. URL: http://dl.acm.org/citation.cfm?id=3305381.3305576.

[11] C. Molnar. *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable*. https://christophm.github.io/interpretable-ml-book/. GitHub, 2019.

[12] F. E. B. Otero and A. A. Freitas. "Improving the Interpretability of Classification Rules Discovered by an Ant Colony Algorithm". In: *Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation*. GECCO '13. Amsterdam, The Netherlands: ACM, 2013, pp. 73–80. ISBN: 978-1-4503-1963-8. DOI: 10.1145/2463372.2463382. URL: http://doi.acm.org/10.1145/2463372.2463382.

[13] T. Peltola. "Local Interpretable Model-agnostic Explanations of Bayesian Predictive Models via Kullback-Leibler Projections". In: *ArXiv* abs/1810.02678 (2018).

[14] E. Raff et al. "An investigation of byte n-gram features for malware classification". In: *Journal of Computer Virology and Hacking Techniques* 14 (2016), pp. 1–20.

[15] M. T. Ribeiro, S. Singh, and C. Guestrin. *"Why Should I Trust You?": Explaining the Predictions of Any Classifier*. 2016. arXiv: 1602.04938 [cs.LG].

[16] R. Ronen et al. *Microsoft Malware Classification Challenge*. 2018. arXiv: 1802.10135 [cs.CR].

[17] A. Saabas. *TreeInterpreter*. 2015. URL: https://github.com/andosa/treeinterpreter.

[18] S. Shirataki and S. Yamaguchi. "A study on interpretability of decision of machine learning". In: *2017 IEEE International Conference on Big Data (Big Data)*. Dec. 2017, pp. 4830–4831. DOI: 10.1109/BigData.2017.8258557.

[19] A. Shrikumar, P. Greenside, and A. Kundaje. "Learning Important Features Through Propagating Activation Differences". In: (2017). arXiv: 1704.02685 [cs.CV].

[20] E. Strumbelj and I. Kononenko. "An Efficient Explanation of Individual Classifications Using Game Theory". In: *J. Mach. Learn. Res.* 11 (Mar. 2010), pp. 1–18. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1756006.1756007.

[21] S. Wachter, B. Mittelstadt, and C. Russell. *Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR*. 2017. arXiv: 1711.00399 [cs.AI].

CHAPTER 5

Interpreting Machine Learning Malware Detectors Which Leverage Convolutional Neural

5.1 Introduction

The significantly increased processing power since the early 2000's not only led to increase use in machine learning but also caused the algorithms being used to grow more complex. Further, the increased availability of large data sets provides the opportunity to learn more complex patterns. This has lead to a significant increase in the ability of machine learning models in image classification and computer vision tasks where architectures such as the Convolutional Neural Network (CNN) can now out preform humans in some scenarios. As discussed earlier, Machine learning has also grown more popular in the malware detection and analysis domain. However, the high performance of machine learning in machine vision has now lead to the adaptation of image classification algorithms, such as the CNN, in the malware detection domain as well. However, with these more complex classification algorithms, it becomes increasingly difficult to understand what exactly a model has learnt.

Machine learning techniques not used for image classification have also been applied to augment both of the traditional malware detection approaches (i.e. static

and dynamic analysis), as we have scene in chap 4. This typically involves a complex feature engineering and extraction phase which has to be fine tuned for different systems and different malware families. This incentivized the use of deep learning in order to automate some portion of the feature extraction process. Further more, architectures which are designed to make use of the ordinal information contained within the input sample have been favoured. This is because the order in which instructions appear in a binary is massively significant in determining its functionality. Thus, recurrent neural networks (RNN) which have typically been used for natural language processing are an obvious candidate. However, RNNs have trouble dealing with long term dependencies within the sample they are classifying. Further, [10] made the observation that malware converted to images belonging to the same family have visual similarities between them and have dissimilarities with malware belonging to different families which can be distinguished by the human eye. This caused some researchers to turn to CNNs as they also take advantage of ordinal information contained in the input data as well as use the spatial information contained in images.

As discussed earlier, the downside to such complex approaches is the resulting models are not easy to understand or lack interpretability. Malware analyst prefer explainable solutions as they must fine tune their systems in order to limit the number of false positives and false negatives. However, if you do not know what the model has learnt, or why it is making a prediction, then it is difficult to make adjustments as you are essentially working with a black box. Further, the inherent risk involved with new technologies means that stake holders must be convinced the model is learning something relevant to the task at hand. This was much easier with traditional approaches where the classification was an easy to understand process, however now it is no longer evident how the model is making a classification. Additionally, a growing problem in malware analysis is the large amounts of data one must sift through to determine the functionality of a malware binary. Patterns the model learnt should be used to help with this issue.

If a fine grained interpretation of a malware classification model can be obtained, one which isolates specific lines of code as significant for a classification of a single

sample (i.e. a local interpretation), then this interpretation can be used to aid in down stream tasks such as highlighting code snippets which significantly contributed to a classification decision. This would give malware analysts a starting point and help limit the amount of time and effort needed to determine the functionality of a malware sample. Further, isolated lines of code which are deemed significant can be used to detect when a model is learning irrelevant relationships. These can then be corrected to decrease false positive and false negative rates. Lastly, if these significant lines of code can be shown to corroborate industry knowledge then this can show the model has learnt something which is relevant and help improve confidence from stakeholders. This would not only put stakeholders minds at ease but would increase industry adoption for this emergent technology.

Thus, in this chapter we focus on augmenting the approach of using a CNN trained on the image representation of malware binaries for static analysis. We do this with the goal of providing a fine grained local interpretation of prediction results while maintaining good classification performance relative to similar models in the literature as well as keeping the simple automated feature extraction from raw data which CNNs provide. We start with a brief review of the application of CNNs to malware classification, we then detail the specifics of our method for generating and classifying malware images and interpreting our classification results. Next we have a discussion of our results and end with conclusions and future work. To the best of our knowledge, the interpretation approach used in this work has not been done before.

5.2 Literature Review

Recently there has been some interest in applying CNNs to malware classification. In [20], they were able to achieve a 96.7% accuracy classifying malicious binaries against benign binaries. This was accomplished by first mapping op code sequences of length 2 from a sample to a 2 dimensional feature map where the value of each "pixel" in the feature map was determined by multiplying the information gain of the corresponding op code sequence in the sample by the probability of said op code

given said sample. Next the resulting "images" where enhanced to create a larger contrast between the malicious and benign samples before applying a CNN to the final image set for classification.

In [19] they converted the first 784 bytes of various network traffic representations into 28×28 grey scale images to train a CNN to detect malicious network traffic and different families of malicious network traffic. With their best preforming representation, their CNN achieved a 100% accuracy when distinguishing between malicious and benign network traffic and a 98.65% accuracy when distinguishing between families of malicious network traffic.

In [8], they were able to classify a data set containing both benign and malicious binaries belonging to 12 different malware families by using a hybrid feed forward-CNN classifier. The feed forward portion of the classifier used features extracted from the PE meta data and imports while the CNN used opcode sequence data where each row of the input volume corresponded to the one hot encoding of an opcode vector. Their architecture was able to achieve a 0.92 f1-score, however the feed forward and CNN alone were able to achieve a 0.90 and 0.91 f1-score respectively while an SVM trained on the same features achieved a 0.92 f1-score, so these results serve more as a proof of concept rather then indicating a superior solution.

As you can see there are various methods used to convert malware samples to input for CNN classifiers. However one popular method put forward in [10] and used in the following works is to convert the binaries to grey scale images by interpreting the raw binary data as a sequence of pixels, where each byte represents the grey scale value of its corresponding pixel in the range [0,255]. The problem with this process is that the resulting images are not of uniform length, thus they must be reshaped in order to match the input dimensions of the CNN classifier.

In [5], the authors used a CNN with alternating convolutional then subsampling layers and several fully connected layers to classify a data set of grey scale images from 25 malware families. Here the input was rescaled to uniform dimensions, losing some information. They also preform image augmentation such as rotation and shifting to reduce overfitting. The resulting model managed to obtained a 94.5% accuracy when

classifying malicious vs. benign samples.

The authors in [18] were able to classify malicious Internet of Things (IoT) malware by converting the binaries in a data set containing 365 samples to grey scale images and then rescaling the images to uniform dimensions to train a CNN with. Half of the samples were IoT malware belonging to two major IoT malware families, and the other half were benign Ubuntu system files. The resulting classifier achieved a 94% accuracy.

Transfer learning was used in [13] by taking the first 49 layers of the ResNet-50 architecture and swapping the last layer for a 25-node softmax layer to make classifications on a data set that contained grey scale images of 25 different families of malware. The images were first converted to RGB since ResNet-50 is designed for 3-channel image input. During training all but the final layer weights were frozen and the classifier was able to obtain an accuracy of 98.62%. Here, the varying size of the malware binaries created varying sized images that were rescaled, still offering good results despite the loss of information.

[3] also used transfer learning on the same data set as the above, except they used the Inception-V1 architecture and froze all but the last fully connected layer and the softmax layer which was replaced with a 25-node softmax layer. Similarly they converted the grey scale images to RGB images by duplicating the grey scale channel three times. Their approach obtained a very impressive 99.25% accuracy.

They also claim to provide an interpretation of the predictions by using LIME [14] to highlight important areas of an input sample. However, the proposed method can only highlight important regions of the input image called "super-pixels" which encompass very many pixels which each map to a byte of code. The regions are of varying size but there are 200 total, meaning that even a modestly sized binary of 200,000 Bytes would have super pixels highlighting on average 1000 bytes of code each. We would like to improve on this approach in order to provide more granular interpretations. Further, [3] also used their approach on the same data set used in this chapter and obtained a 98.13% accuracy. We will return later to these results for comparison between our methods.

Additionally, [6] converts the same data set used in this chapter into grey scale images and trains a CNN classifier without the use of transfer learning after down sampling the images to unifrom size. The classifier has 3 convolutional layers followed by a fully connected layer and a softmax classification layer and was able to achieve a 97.5% accuracy. We will return also to these results for comparison between our method with a trained from scratch method.

There has been a plethora of papers published with differing techniques used to interpret or visualize what machine learning models and neural networks have learnt. However, for the sake of brevity, we will discuss some of the techniques used for convolutional neural networks only.

Layer-wise Relevance Propagation (LRP), described in [2] as a set of constraints, is used to visualize where the model is placing its emphasis when making classifications by back propagating a models prediction using its weights and some decomposition function which returns the relevance of previous nodes to that prediction. The constraints ensure that the total relevance is preserved from one layer to the next as well as that the relevance of each node is equal to the sum of relevance contributions from its input nodes which in turn is equal to the sum of relevance contributions to its output nodes. Any decomposition function following these constraints is considered a type of LRP.

In [17], they propose DeepLIFT which, in contrast to LRP, attributes to each node a contribution to the difference in prediction from a reference prediction. DeepLIFT back propagates just this relative difference in prediction scaled by the difference in intermediate and initial inputs.

The authors in [14] put forward Local Interpretable Model-agnostic Explanations (LIME), which was used in [3] above, to explain predictions using an approach which trains an interpretable classifier by heavily weighing samples nearer to a sample of interest in order to locally approximate the non-interpretable or black-box model. This work was extended by Tomi Peltola in [11] to generate local interpretable probabilistic models by minimizing the Kullback-Leibler divergence of the predictive model and the interpretable model in order to provide explanations that account for model

uncertainty.

There have also been several implementations of the above methods. The creators of LIME developed a python library available at `https://github.com/marcotcr/lime`. Additionaly, there is the iNNvestigate [1] python library available at `https://github.com/albermax/innvestigate` which was used in this chapter and provides implementations of LRP as well as many other useful interpretation methods met for convolutional neural networks, although many of them can be applied to other types of neural networks not working with image data.

5.3 Method

Training and classification were done on a subset from a data set of 10,896 malware files belonging to 9 different malware families.[1] The data set is discussed in [15]. Each sample consists of the hexadecimal representation of the malware's binary content in a .bytes file as well as its corresponding assembly code in an .asm file. The hexadecimal representations were preprocessed as followed. First, we determined the total length of each binary. Since our goal is to provide a fine grain interpretation, we wanted to avoid resizing images to fit the input of the CNN if the resizing caused information loss. Thus, we deiced to only scale up images by padding them with zeros, rather then scaling them down. This is because when we scale down an image the resulting image's pixels will actually map to more then one pixel in the original, and therefore more then one byte of code. Therefore, even if we obtain the importance of a single pixel of the rescaled input image, we still do not have a fine grained approach, since the mapping from rescaled input image to full sized image and then to the bytes and finally the assembly code will be a one-to-many mapping, which is increasingly so with large binaries that require more drastic rescaling. So, we picked a size range where the majority of the binaries resided, that is the range of 101,400 to 200,934 bytes. The files which were less then 200,934 bytes in length were padded with bytes of all 0's before and after the binary so that all the binaries were the same size.

[1] The data set was downloaded from `www.kaggle.com/c/malware-classification/data`

The padding before and after the binaries was equal so that the actual binary was centred vertically in the image, although it took up the entire width of the image. The resulting data set contained 2,114 binaries with 6 classes total. The class details are summed up in table 5.3.1.

Table 5.3.1: Class distribution in Data Set

Class No.	Family	Samples	Type
1	Ramnit	635	Worm
2	Lollipop	68	Adware
4	Vundo	188	Trojan
6	Tracur	145	TrojanDownloader
8	Obfuscator.ACY	810	obfuscated malware
9	Gatak	268	Backdoor

Next, the binaries where converted into image tensors of the shape $(183, 183, 6)$. This was done by placing the first 6 bytes of the binary in the input tensor positions at location (0,0,0) through (0,0,5) the next six bytes in the tensor positions at location (0,1,0) through (0,1,5), and so on, until all the bytes were processed. The reason 6 was chosen as the number of channels was because [12], which examined the effectiveness of different lengths of byte-grams for malware classification, found that 6-byte-grams were the most effective compared to other lengths of byte-grams. A byte-gram is as a sequence of bytes which appear consecutively in a binary. These are analogous to n-grams which are a sequence of n words or characters which appear in text. The intuition here was that each pixel, which contains the 6-byte-gram in the pixel's 6 channels, would contain what could be thought of as a "byte word" leaving the filters to learn to detect significant byte words and in later layers significant sequences of byte words.

To the best of our knowledge, this the first time someone has applied a CNN to a malware binary classification task where the binaries were converted to "images" with 6 channels without the use of downsizing. The 6-channel input has an added benefit of fitting more information into a smaller volume, this means we can have a compact

input volume which can still contain all the information of a 200,934 byte binary, therefore helping in our task of creating fine grain interpretations. The other two dimensions were set to $\sqrt{200,934 \div 6} = 183$, since this created a square image, which is what the models we are using for comparison (discussed in section 5.2) also used. Figure 5.3.1 shows the resulting images of two samples for each of the 6 classes. The images in the column labeled F are the first three channels of the samples interpreted as RGB channels while the images in the columns labeled L are the last three channels of the samples interpreted as RGB channels. As you can see there is some similarity between images of the same classes and greater difference between images of different classes. Additionally, there is a lot of similarity between the last and first three channels. However, the CNN model is indifferent to the number of colour channels or human detectable features and can find structure, imperceptible to humans, in an arbitrary number of channels, therefore we must wait until the classification results before we can draw any conclusions.

The neural network starts with 2 blocks of the classic convolution, ReLU, MaxPool architecture. This architecture was chosen as it has been shown to work well in the literature and is also similar to what was used by the models used for comparison. We went with just 2 blocks as we wanted to limit the number of parameters given the small size of our data set. These 2 blocks were followed by a single fully connected layer with 512 neurons with the ReLU activation function then a softmax classification layer with 6 neurons. 512 neurons were used as experimentation showed this number to work best on the validation set despite adding a large number of parameters and the possibility of over fitting. The two convolutional layers used 128, then 256, 5x5 filters with strides of 2 and the MaxPool layers used 2x2 pool size with strides of 2. This was done mainly because the small data set size meant we had to shrink the volume as fast as possible in order not have too many layers or too many neurons in the last volume before the first dense layer thus keeping the number of parameters low. To reduce over fitting dropout with a rate of 0.4 was also used on the connections between the last MaxPool layer and the first fully connected layer since these connections accounted for 13,107,712 of the 13,949,575 trainable parameters. Further, L2 regularization was

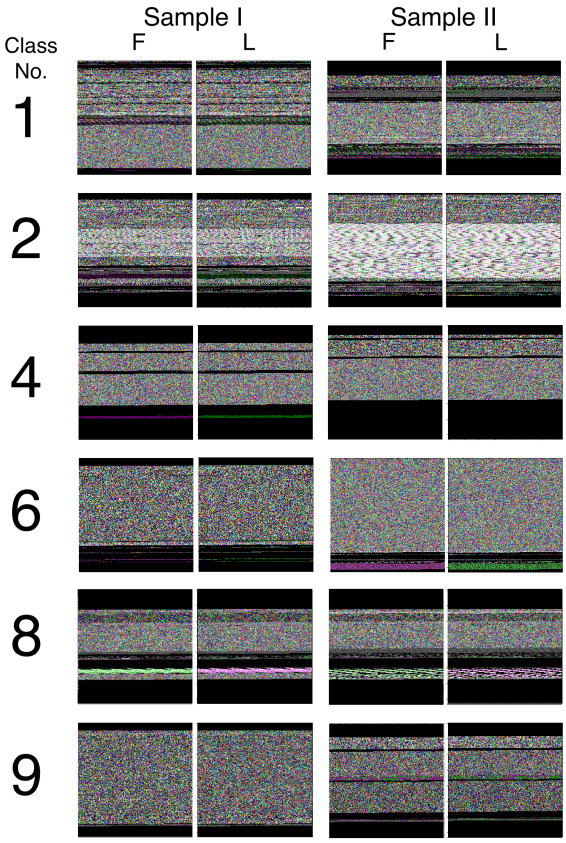

Fig. 5.3.1: Images from two samples from each of the 6 classes. Images in columns labeled F are the first three channels of the samples interpreted as RGBchannels while the images in columns labeled L are the last three channels

used with a 0.1 penalty on both convolutional layer weights and the weights between the last MaxPool layer and the first fully connected layer. Figure 5.3.2 shows a summary of the architecture of the CNN used. The figure was created using software available online at `http://alexlenail.me/NN-SVG/LeNet.html` which is described in [9].

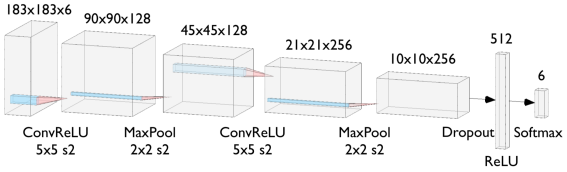

Fig. 5.3.2: Model Architecture

The model was trained as follows. First the data set of 2,114 samples was randomly split into 3 disjoint sets, the training set, validation set, and the test set, each with

approximately the same class distribution. The Training set had 1,514 samples, the validation set 100 samples, and the test set had 500 samples. The validation set was used to tune the model's hyper parameters and needed to be small to ensure the test set was large enough to be significant for evaluation while the training set was large enough for the model to learn generalized patterns despite the small size of the total data set. The test set was not used except at the end of training in order to evaluate the final model. The training was done using the Adam optimizer with 256 batch size over 80 epochs. Classes were weighted inversely proportionately to the class size in order to account for class imbalances. The model was implemented and trained using the Keras [4] python library.

5.4 Results

5.4.1 Evaluation of Our Model

After training the model and using the weights with the lowest validation loss over the 80 epochs the model obtained a 98.1% balanced categorical accuracy and a 0.237595 categorical crossentropy loss on the left out test set. Balanced accuracy was used since there was a class imbalance in the data set. Further, the model does not seem to suffer from over fitted as indicated in figure 5.4.1 which shows the test and validation loss history plotted against the number of epochs. This is also evident from figure 5.4.2 which shows the test and validation categorical accuracy plotted against the number of epochs.

5.4.2 Interpretation

After training, a process called Layer-wise Relevance Propagation (LRP) [2], which returns the relevances of all input nodes to a sample's prediction, was used to find important input pixels. In our experiment we used the iNNvestigate [1] python library's LRP implementation. Once we had the relevances of each input node we averaged them across the 6 channels to get a 2D relevance map. For visualization,

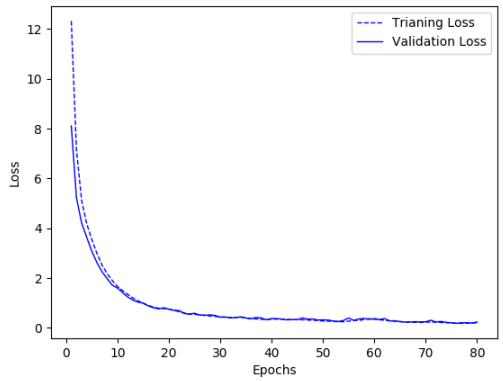

Fig. 5.4.1: Test and Validation Loss History

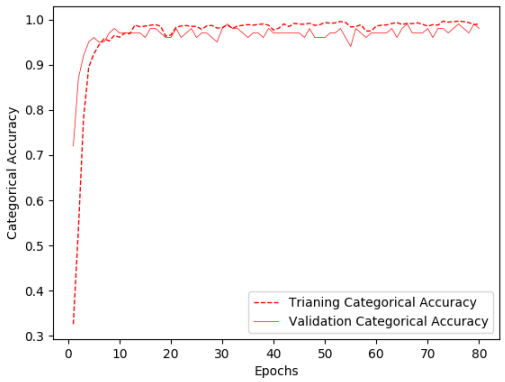

Fig. 5.4.2: Test and Validation Balanced Categorical Accuracy History

we used the seismic colour map from matplotlib [7] to plot the relevance map. Figure 5.4.3 shows the image resulting from taking the first 3 channels of the sample associated with the binary with ID 0AnoOZDNbPXIr2MRBSCJ, and the image resulting from taking the last 3 channels of the same sample, as well as its corresponding relevance map. The pixels highlighted with red contributed positively to the models prediction while the pixels highlighted in blue contributed negatively. Sample 0AnoOZDNbPXIr2MRBSCJ was correctly classified by our model with a 99.99% chance of belonging to class 1.

Although we cannot obtain a lot of specific information by looking at these images and the LRP visualization, we can still obtain some broad insight into the models prediction. As you can see from figure 5.4.3, the classifier recognized a large set

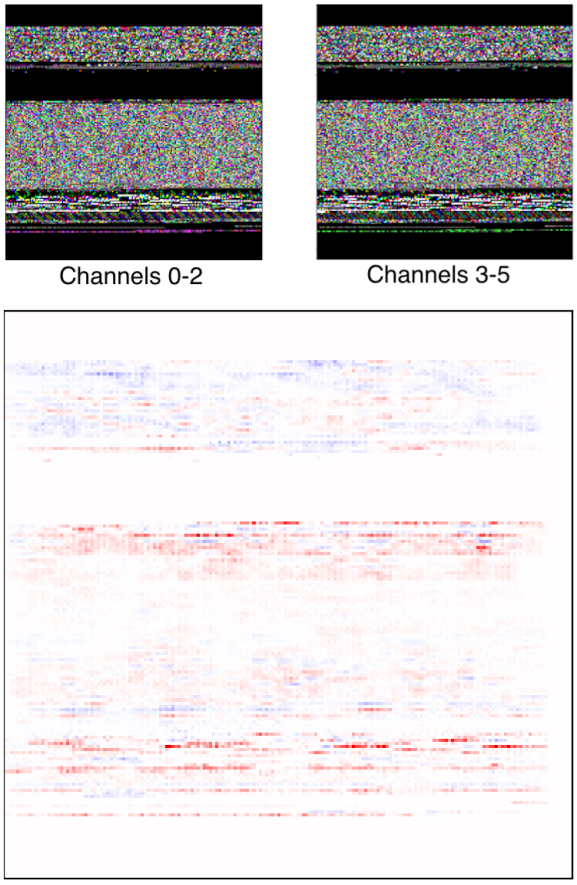

Fig. 5.4.3: 0AnoOZDNbPXIr2MRBSCJ Relevance Map

of pixels as an indication that this sample belonged to class 1. Further, there was still some pixels which contributed negatively to this prediction, mostly in the upper portion of the image above the band of white which corresponds with bytes of 0's. This suggest that the portion of the code containing functionality related to class 1 in located in the lower portion of the binary and the upper portion is mostly related to other classes or is benign. However, these statements are always dependent on how well the classifier was trained to learn relevant information. Further, if the relevance was intensely focused on a few small areas we could be worried that the model is relying on a small set of features to make classifications, meaning small changes could lead to misclassifications, and this can be taken advantage of by malware authors.

To obtain a fine grained interpretation such as highlighting specific lines of code, we obtain a list of indices of the input pixels sorted by their relevance. We then move down the list in descending order of relevance and obtain the most relevant 6-grams

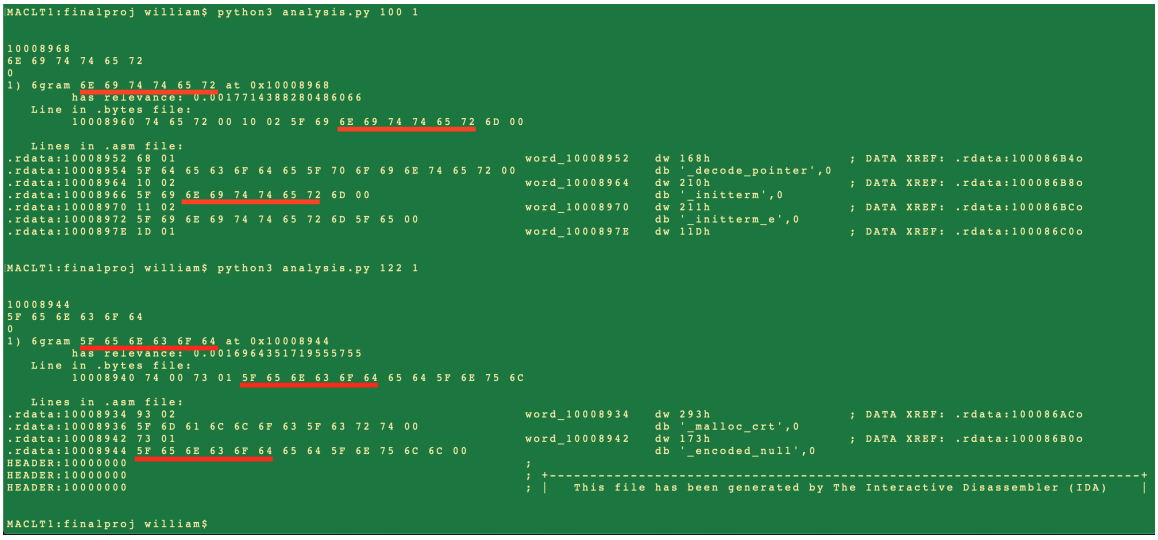

Fig. 5.4.4: Terminal output when working backwards from the relevant 6-grams to the code snippets

for interpretation first. The relevant 6-gram's position in the sample's input tensor, the input tensor itself, and the corresponding byte file are needed to determine the 6-gram's address in the assembly code. These are needed to determine the amount of padding to account for, as well as the starting address in the byte file, since they don't all start at the same number. However, in a application scenario all of this information would be available. Once we have the exact address in the assembly code it is a trivial task to find the lines of Assembly code which contain the relevant 6-gram.

It should be noted that finding the exact lines of code which contributed to a prediction, as well as their exact ordering relative to the size of their contribution, is very difficult, if not impossible with most other model architectures. For example, if n-gram analysis was used, where the presence or frequency of an n-gram is used as a feature, then even though we may have the contribution of each n-gram feature, we do not know which occurrence of said n-gram in the binary contributed the most. This is true in some way for most frequency based techniques. Further, for other CNN based techniques, we have the problem of rescaling causing the contribution of one input node being distributed across many pixels in the original image. It is only

when positional information of each 6-gram is maintained from raw binary to input tensor, as we have done so here, where we are able to so easily make such precise interpretations of the model.

Figure 5.4.4 shows the terminal output when working backwards from the relevant 6-grams to the code snippets in the assembly code. Note that some of the assembly code has been truncated so we could not find the corresponding code snippets to many of the most significant 6-grams but we have done so for the 100^{th} and 122^{nd} most significant. This is not a problem however as typically you would have the full assembly code.

The significance of finding these code snippets which contributed heavily to a prediction is large. For example, there is the case where the code snippets are completely irrelevant to classifying binaries according to functionality despite the model achieving good classification results. In this situation, there is the likely culprit of an incomplete or non-representative data set. It could be that one class has irrelevant but frequently occurring code-snippets that by chance do not appear in the other classes. Here, gathering a larger data set, or even augmenting the current data set so that other classes also include this irrelevant code snippet, can force the model to learn different patterns that exclude this irrelevant feature, which should also improve generalization performance. If the classification performance drops after this, then this could hint at poor feature engineering, since the remaining representative features no longer help the model make predictions. In the case of CNNs applied to images of binaries, this could mean a deeper model architecture that can create more abstract hidden features might be needed, or that the representation of binaries as images themselves is unhelpful.

In the case where code snippets with high relevances to the model's predictions are known to be relevant to classifying binaries based off their functionality, then the results of the model are in a way, validated. As we said earlier, this can help malware analysts as they can be shown where to start their static analysis, as well as help stakeholders feel confident in the black-box CNN model.

5.4.3 Comparison With Other Models

Table 5.4.1 shows the confusion matrix of our model on the left out test set where we can see the model preformed well for all classes. Table 5.4.2 and 5.4.3 show the confusion matrix of the models used in [3] and [6] respectively (both are discussed in section 5.2). The columns and rows for classes 3, 5, and 7, which were not used in our experiment, have been omitted.

Table 5.4.1: Confusion Matrix for Our Model on the Left Out Test Set

Class No.	1	2	4	6	8	9
1	**149**	0	0	**1**	0	0
2	0	**16**	0	0	0	0
4	**1**	0	**44**	0	0	0
6	0	0	0	**32**	**1**	**1**
8	**2**	0	0	0	**190**	0
9	**1**	0	0	0	0	**62**

Table 5.4.2: Confusion Matrix for Model used in [3]*

Class No.	1	2	4	6	8	9
1	**154**	0	0	0	**3**	0
2	0	**238**	0	0	**3**	**1**
4	**1**	0	**33**	**1**	0	0
6	**1**	0	0	**63**	**1**	0
8	**2**	0	0	0	**119**	0
9	0	**4**	0	0	0	**102**

*columns and rows of classes classes

3,5, and 7 have been omitted

Table 5.4.3: Confusion Matrix for Model used in [6]*

Class No.	1	2	4	6	8	9
1	**1490**	**4**	**2**	**9**	**28**	**3**
2	**6**	**2440**	0	**7**	**8**	**16**
4	**3**	0	**461**	**1**	**3**	**2**
6	**8**	**6**	**2**	**713**	**10**	**9**
8	**44**	**4**	**8**	**17**	**1138**	**8**
9	**2**	**2**	0	**6**	**5**	**996**

*columns and rows of classes classes

3,5, and 7 have been omitted

Using these confusion matrices to calculate the balanced accuracy score of each model, we can get a decent comparison. However, the reader should note that the other models were trained on more classes and with much more training data so these are not perfect direct comparisons of our approaches. Table 5.4.4 sums up the comparisons between the three models. As you can see, we score competitive balanced accuracy score with a very light weight model. Further, we are able to give a fine grained analysis of our predictions using the method detailed in section 5.4.2 and this method cannot be easily applied to the other models without added processing and sacrificing the preciseness of our method. This is due the decision to not rescale the images, meaning we can map one relevant input node to exactly one 6-gram in the binary and then to the corresponding assembly code.

Our model does have draw backs however. Unlike the other models ours is only designed to work with samples in a specific size range and therefore one would need multiple models in order to achieve the same effect across different size ranges. One possible solution would be to train on a data set where the samples in the appropriate size range are not rescaled but padded like we did here, and the images which are too large are rescaled to fit. This however would mean the interpretation method would

only maintain its fine granularity when classifying samples which were not resized.

Table 5.4.4: Model Comparisons

Model	**Balanced Acc.**	**Size**
Our Model	98.1%	2 conv, 1 Dense
Model from [3]	97.04%*	20+ layers see [16]
Model from [6]	96.8%*	3 conv, 1 Dense

Model	**Interpretation**	**Sample Size**
Our Model	Fine grained & precise	104k-200k Bytes
Model from [3]	Broad & imprecise	any size
Model from [6]	none given	any size

*calculated by omitting columns and rows of confusion matrix for classes 3,5, and 7

5.5 Conclusion

In summary, we were able to obtain competitive classification results on a subset of a classic benchmark data set. Compared to other methods we made appropriate trade offs in terms of broad applicability of our model (in that it only works for malware in a specific size range) in return for large gains in interpretability. We thus have provided a proof of concept for 6-channel image based malware classification using a simple convolutional neural network that did not suffer from excessive overfitting

despite a small data set. To the best of our knowledge, this the first time someone has been able to interpret a CNN based malware detector with the granularity which has been achieved here, by applying CNNs to a malware converted to images with more then a single channel in order to avoid rescaling of the image and information loss.

For future work, there is much work to be done in order to better handle binaries of different sizes. If more sophisticated approaches for dealing with binaries of different sizes are implemented, which do not result in information loss, then fine grained interpretations, in the manner we have done so here, can be possible for any malware file. Further, the data set will not shrink as a result of not considering files which are too large. This means more advanced models can be deployed without over fitting the data set, potentially increasing the models performance.

Another interesting possibility is to explore is the application of our approach to graph convolutional neural networks (GCNNs), which are CNNs applied to graph representations, trained on malware classification. It is a popular approach in malware analysis to represent the behaviour or other features of a malware binary in a graph which in many cases will contain direct and explicit functional information. If a GCNN is trained on a dataset where each sample is one of these graph representations of a malware binary, then less time can be spent worrying about weather the model learnt to use features indicative of functionality and translating those features to functionality afterward for interpretation. Instead interpretability can be used to directly make statements about the relevant functionality which the model is perceiving within the sample to make its prediction.

References

[1] M. Alber et al. *iNNvestigate neural networks!* 2018. arXiv: `1808.04260 [cs.LG]`.

[2] S. Bach et al. "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation". In: *PloS one*. 2015.

[3] L. Chen. *Deep Transfer Learning for Static Malware Classification*. 2018. arXiv: 1812.07606 [cs.LG].

[4] F. Chollet et al. *Keras*. https://keras.io. 2015.

[5] Z. Cui et al. "Detection of Malicious Code Variants Based on Deep Learning". In: *IEEE Transactions on Industrial Informatics* 14.7 (July 2018), pp. 3187–3196. ISSN: 1941-0050. DOI: 10.1109/TII.2018.2822680.

[6] D. Gibert et al. "Using convolutional neural networks for classification of malware represented as images". In: *Journal of Computer Virology and Hacking Techniques* 15.1 (Mar. 2019), pp. 15–28. ISSN: 2263-8733. DOI: 10.1007/s11416-018-0323-0. URL: https://doi.org/10.1007/s11416-018-0323-0.

[7] J. D. Hunter. "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3 (2007), pp. 90–95. DOI: 10.1109/MCSE.2007.55.

[8] B. Kolosnjaji et al. "Empowering convolutional networks for malware classification and analysis". In: *2017 International Joint Conference on Neural Networks (IJCNN)*. May 2017, pp. 3838–3845. DOI: 10.1109/IJCNN.2017.7966340.

[9] A. LeNail. "NN-SVG: Publication-Ready Neural Network Architecture Schematics". In: *Journal of Open Source Software* 4 (Jan. 2019), p. 747. DOI: 10.21105/joss.00747.

[10] L. Nataraj et al. "Malware Images: Visualization and Automatic Classification". In: (July 2011). DOI: 10.1145/2016904.2016908.

[11] T. Peltola. "Local Interpretable Model-agnostic Explanations of Bayesian Predictive Models via Kullback-Leibler Projections". In: *ArXiv* abs/1810.02678 (2018).

[12] E. Raff et al. "An investigation of byte n-gram features for malware classification". In: *Journal of Computer Virology and Hacking Techniques* 14 (2016), pp. 1–20.

[13] E. Rezende et al. "Malicious Software Classification Using Transfer Learning of ResNet-50 Deep Neural Network". In: Dec. 2017. DOI: 10.1109/ICMLA.2017.00-19.

[14] M. T. Ribeiro, S. Singh, and C. Guestrin. ""Why Should I Trust You?": Explaining the Predictions of Any Classifier". In: *CoRR* abs/1602.04938 (2016). arXiv: 1602.04938. URL: http://arxiv.org/abs/1602.04938.

[15] R. Ronen et al. *Microsoft Malware Classification Challenge*. 2018. arXiv: 1802.10135 [cs.CR].

[16] Christian S. et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV].

[17] A. Shrikumar, P. Greenside, and A. Kundaje. "Learning Important Features Through Propagating Activation Differences". In: (2017). arXiv: 1704.02685 [cs.CV].

[18] J. Su et al. "Lightweight Classification of IoT Malware Based on Image Recognition". In: *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 02. July 2018, pp. 664–669. DOI: 10.1109/COMPSAC.2018.10315.

[19] W. Wang et al. "Malware traffic classification using convolutional neural network for representation learning". In: *2017 International Conference on Information Networking (ICOIN)*. Jan. 2017, pp. 712–717. DOI: 10.1109/ICOIN.2017.7899588.

[20] J. Zhang et al. "IRMD: Malware Variant Detection Using Opcode Image Recognition". In: *2016 IEEE 22nd International Conference on Parallel and Distributed Systems (ICPADS)*. Dec. 2016, pp. 1175–1180. DOI: 10.1109/ICPADS.2016.0155.

CHAPTER 6

Robustness Metric

6.1 Introduction

In malware detection a misclassification can cause huge delays if a benign file which is needed for some time sensitive task is flagged as malicious. Furthermore, there is the risk that a malicious file is permitted to execute causing unforeseen damage to the users system. Since a small to large business may encounter thousands to millions of benign files each day, a hard requirement of malware detection systems relied upon by large companies is a low false positive rate (FPR). Due to the large volume of files, even a FPR of one in a thousand would lead to daily stoppages and false alarms. Furthermore, any fragility in the models accuracy can be taken advantage of by malware authors looking to bypass detection. It is clear that misclassification in this scenario carries a heavy risk and to quantify the rate of misclassification, metrics such as FPR or accuracy are typically used. However, these metrics do not capture the full story when trying to convey how robust a detection system is to the anti detection efforts of malware authors. A contrived example would be a model that learnt to associate a few superficial features strongly with the property of being benign but none the less has high accuracies on held out test sets. Here, superficial is taken to mean that the features are not actually indicative of being benign but due to peculiarities in the model's training set, the model treats them as such. In such a case, if a malware author was to learn of this association, they could easily change

their binaries to bypass the detection system, without having to change the malicious functionality of their malware. This means that despite the high accuracy, the model is still not robust to anti detection efforts.

A metric that better illustrates the robustness of a classification model is needed. This metric should quantify the accuracy in relation to the number of features which are no longer following the model's learnt relationships. This is because we want to measure how well the model preforms as features are made to no longer have their expected values given a samples actual class and thus no longer aid in prediction. In this chapter we refer to such features as "deactivated". Further, the metric should be concerned only with the inputs and outputs of the model. This allows the model to be treated as a black box so that the metric can be applied to any model, that is the metric will be model agnostic. Additionally, the metric should have some degree of customization. This is because not all models are the same. They use different types of features with different valid feature ranges and are applied in varying domains. This means that what it is for a feature to be "deactivated" will be largely different among different applications and scenarios and therefore will have to be determined and asserted by the user of the metric. Further, the technique for calculating the metric should allow the user to determine which set of features deactivation caused the accuracy to drop below some minimum performance requirement. This follows from the fact that not all models use the same features, thus a model may only need a few features to be deactivated for its performance to drop significantly, however if the features use by that model are harder to change without removing the malicious properties of the malware, then this model is still robust.

In this chapter we propose a new robustness metric inspired by area under curve (AUC) which meets these requirements. We start with a discussion of how the metric is calculated and why the metric is calculated in this way. We then use the metric on several classification models that have been trained on a data set of features extracted from benign and malicious binaries. Finally we end with a discussion of our results and possible future work.

6.2 The Robustness Metric

The metric is calculated as follows. First, a numerical measure of feature significance must be found for each feature. For different models this means different things. In the case of a logistic or linear regression classifier, the feature significances is equal to the coefficient or weight of the feature. For decision trees and random forest it is the gini importance and average gini importance among the constituent trees respectively. In the case of neural networks we use layer-wise relevance propagation [1] to obtain the relevance of each input node for each sample. We then average the absolute value of these relevances in order to get the average relevance of each node. This value is used as the feature significance of the node's corresponding feature. The average absolute value is used because for one sample a node may have a large negative contribution and for another sample, a large positive contribution. If the actual relevance value is used, then during averaging these relevances will cancel out despite the associated feature having a large effect on both predictions. For other models there are typically already standard accepted practices for obtaining a feature's significance but in the case that there is not, a method known as permutation importance [2] can be used. In this case, classification on a set of samples is done before and after a feature is permuted. The increase in error is what determines the importance of said feature.

After the feature significance is found, the model's balanced accuracy on a test set is found. A test set is used for the same reason you do not report a models accuracy on the training set. Balanced accuracy is used because it is calculated by taking the weighted average accuracy for each class, where the weights are inversely proportionate to the class frequency. This means that a random classifier is expected to get 50% balanced accuracy even on imbalanced data sets. This allows for direct comparison of the robustness metric when evaluated on test sets with varying class distributions. Next the most significant feature is deactivated and the balanced accuracy with respect to the same test set is determined again. This process is repeated, deactivating the next most significant feature each time. The features are deactivated in order of significance because if a feature is not very important then we are not interested

in how well the model preforms after its deactivation since a good performance after removing irrelevant information is not surprising or noteworthy. The way in which features are deactivated is discussed at the end of this section.

After this we have a list of balanced accuracies $\{a_0, a_2, ..., a_{n-1}\}$ where a_k is the balanced accuracy of the model when the k most significant features are deactivated. We then plot these accuracies along the y-axis with the number of deactivated features on the x-axis to obtain a curve. After this, we take the area under the curve and we divide it by the number of features there are, then subtract 0.5 and multiply it by 2. The result is the value of the robustness metric.

The area is taken because it captures the total accuracy across each iteration of deactivating a feature then evaluating the model. The area is divided by the number of features because the balanced accuracy, which is on the y-axis, is always between 0 and 1. Thus, if the area is taken between 0 and the number of features, n, then the area is always between $0 \times n$ and $1 \times n$. So dividing by n means the area is scaled between 0 and 1. The scaled area is corrected by subtracting 0.5 because a random classifier is expected to have a 0.5 accuracy thus half of the area is not actually attributable to the model. In some rare cases the balanced accuracy may dip below 0.5 enough that the scaled area is below 0.5, meaning the corrected scaled area would be negative. In this case, the scaled corrected area is set to 0. This means that the scaled and corrected area is now in between 0 and 0.5, so it is then multiplied by 2 so that it is between 0 and 1, giving us our final robustness value.

For deactivating features there are several possible methods. A naive approach, hence referred to as the *zeros method*, would be to set the feature's values to 0. However, this may not correspond to a realistic value for all features and can therefore give untrustworthy results. Another approach, which we will refer to as the *random permutation method*, would be to randomly permute the feature column. This approach is inspired by the permutation importance method discussed early. Another approach could be to use a reference sample and set deactivated features equal to the corresponding feature value in the reference sample. The reference sample can be hand picked to be an average of some class of interest or some other value. We will

hence forth refer to this method as the *reference method*.

This gives the user of the metric a good way of customizing it for their specific use case. For example they may have a mix of binary features and numerical features and they may want to set the binary features to 0 and the numerical features to their average value when deactivating them. In any case the user can define exactly how each specific feature is deactivated rather then doing the same for all of them. In the following section, we test all three of these approaches. The reference sample is set to be the average of the benign class. This was done because we are typically concerned with detecting malicious samples when classifying binaries, so setting a feature to its average value within the benign class would isolate the effect that the remaining features had on obtaining a malicious classification.

The reader should note that sometimes we are more concerned with FPR as appose to accuracy. In this case one can replace the balanced accuracy at each step with the FPR and get a similar robustness metric which is conditioned on FPR rather then accuracy. The same is true for other accuracy like metrics as well.

6.3 Method

In our experiment we first trained a neural network, a decision tree, random forest, and a logistic regression model on a data set containing 77 features extracted from malicious and benign binaries. The data set contained 14599 malicious samples and 5012 benign ones. The data set was split into a train and test set containing 17649 and 1962 samples respectively, with equal class distributions. The features were also all scaled using min-max scaling. The neural network had 77 input nodes, two hidden layers with 100 and 30 nodes respectively, and an output layer with 2 nodes, one for each class. Once the training was complete, the models were evaluated on a test set. The neural network obtained a balanced accuracy of 96.16%, the random forest a balanced accuracy of 98.44%, the decision tree 98.26% and the logistic regression model a balanced accuracy of 94.10%. Next we obtained the robustness metric for the four models using the process detailed in section 6.2, using the test set to determine

the balanced accuracy at each step. Table 6.3.1 shows the results.

Table 6.3.1: Robustness Metrics For Trained Models

Model	Zeros	Rand. Perm.	Ref. Sample
Neural Network	0.1236	0.0896	0.0973
Random Forest	0.0499	0.0779	0.0880
Decision Tree	0.0543	0.0546	0.0798
Logistic Reg.	0.0000	0.0519	0.0279

As shown, the neural network consistently out preforms the other models in terms of robustness regardless of which method is used for deactivating the features. This is expected since the neural network has 130 intermediate nodes across two layers each of which can learn a different relationship between the features and the predicted value. This means that if a feature is deactivated, then the nodes which learnt relationships not relying on said feature can still produce correct predictions.

The random forest got second except for in the case of the zeros method where it got third. This makes sense as the random forest has many different constituent classifiers and much like the neural network, when one feature is deactivated, the trees that do not rely on that feature can produce a correct prediction. In the case of both the random forest and neural networks, redundancies were able to provide more robustness, as expected.

The decision tree got third except for in the case of the zeros method where it got second. Further, the logistic regression classifier consistently got last. This is to be expected as the logistic regression classifier simply takes the sigmoid of the features weighted sum and in has no redundancies to deal with misbehaving features which are heavily weighted. Further, the decision tree only has a single path from the root to each of its leaves and thus cannot correct if the prediction is lead askew by misbehaving features.

The resulting plots when finding the robustness measures are shown in figure 6.3.1

in which you can see the zeros method was much less stable with many spikes in performance which may be misleading as these spikes are the effects of highly unrealistic feature values. This is the suspected reason the random forest obtained worse robustness using the zeros method. The other methods were much more stable with a steady decline and are therefore the recommended choice. The random permutation method can be used to test the models robustness against random chance while the reference method can be used to test its robustness against samples intentionally designed to be misleading, such as adversarial malware.

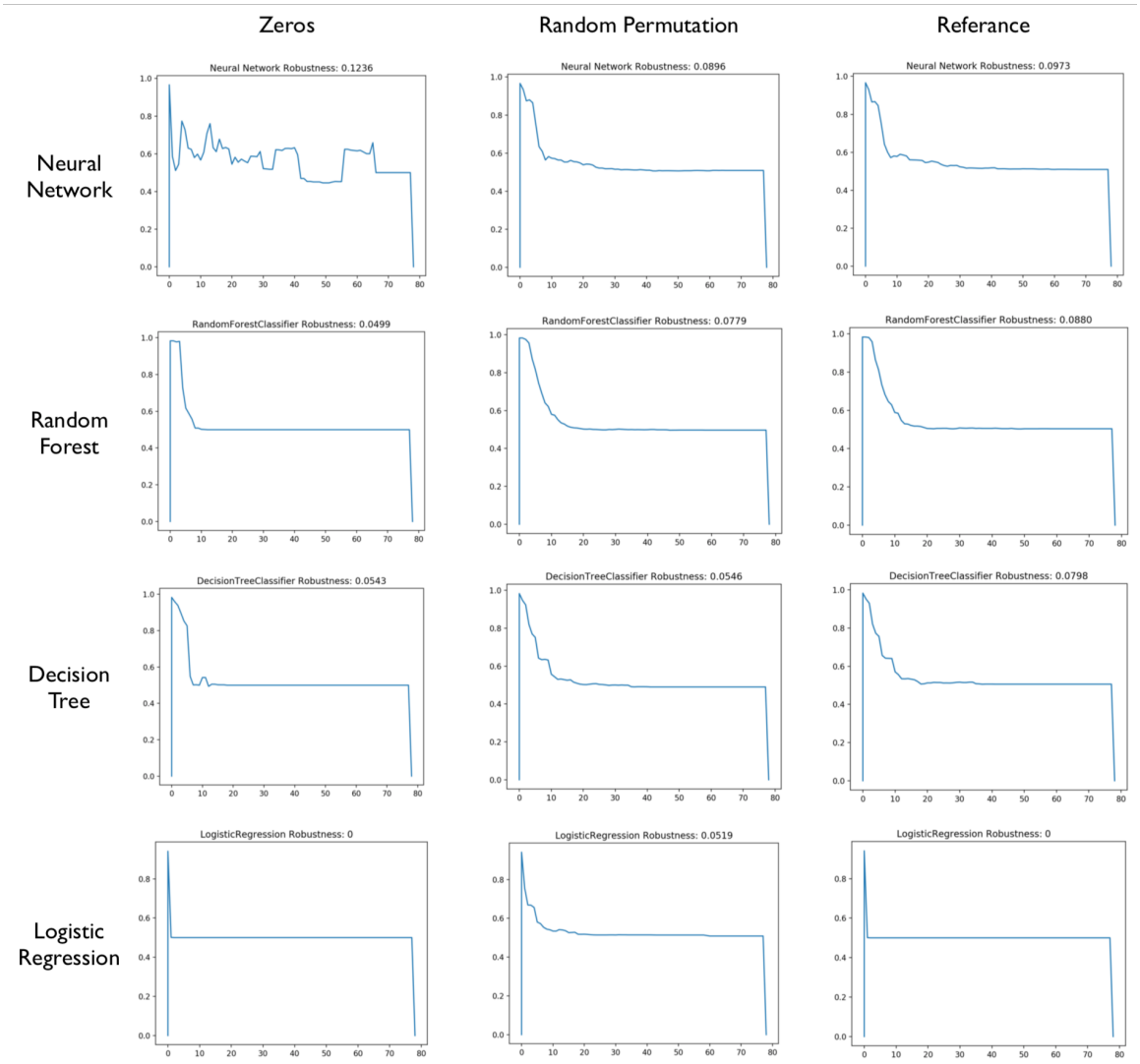

Fig. 6.3.1: Robustness plots with balanced accuracy on the y-axis and number of features deactivated on the x-axis

6.4 Conclusion

In summary, we were able to define an effective algorithm for calculating a robustness metric which met our stated requirements in section 6.1. That is, the metric is defined in terms of deactivated features, it is model-agnostic as it is only concerned with the inputs and outputs of the model, and it can be customized, meaning it can be used meaningfully with many different models, feature types, and use cases. Additionally it is always between 0 and 1 with 0 representing the negative extreme of a random classifier that always scores 50% balanced accuracy and 1 representing the positive extreme of am omniscient classifier that makes perfect classifications, even with no input. Further more, it automatically accounts for class imbalances with the use of balanced accuracy.

Our results also validated our approach since the models which are typically associated with indifference to small changes in input (Neural Network and Random Forest), and who have theoretical support for higher robustness through the redundancy present in their design, scored better then those which are typically sensitive to small changes in input (Decision Tree and Logistic Regression).

For future work, experiments with multi-class models can be conducted. The same method may work for models which return a single set of feature significances even in the multi-class case, such as decision trees, random forests, and neural networks. However in the case of logistic regression, there are a separate set of feature significances for each class if it is a one-vs-rest scheme. In this case it is not obvious if it is better to average the feature significances to determine which order to deactivate the features, or to produce a robustness score for each binary classifier, and then average these robustness scores. Lastly, a similar method needs to be implemented for regression models, ideally one whose value can be directly compared to the value of the robustness metric defined here. Potentially a plot of the mean squared error (instead of accuracy) the classifier achieves on the test set as features are deactivated could be used. In this case a smaller area under the plot would be ideal, as this would be associated with lower mean squared errors as more features were deactivated.

References

[1] S. Bach et al. “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation”. In: *PloS one*. 2015.

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

CHAPTER 7

Conclusion

The work presented in this thesis provided a exploratory overview of machine learning interpretability in the malware detection domain. Starting with Chapter 1, a brief overview of machine learning and some of the popular algorithms therein was provided, as well as an introduction to machine learning interpretability.

In Chapter 2, a literature review concerning the application of machine learning to malware analysis was provided, as well as a discussion of the current strengths and weakness of the machine learning based malware detection approaches in their present state. Emerging threats in the malware domain were discussed, such as fileless malware and unconventional computing paradigms, as well as the practical challenges for machine learning based malware detectors, namely, the cost of training detectors, adversarial malware, and detector interpretability. Lastly, a discussion of possible solutions to these issues was also presented.

In Chapter 3, a Proof of Concept fileless malware, JSLess, was described thoroughly, then implemented, and finally tested against various malware detection software in order to showcase the severity of the threat posed by fileless malware and the necessity for machine learning based malware detectors in the present day. For future work, the functionality for JSLess can be extended, an approach for detecting malware similar to JSLesss can be researched, and the threat of fileless malware in unconventional computing paradigms can be explored.

In Chapter 4, we provided a description of interpretability goals for machine learning based malware detectors and presented techniques for achieving these inter-

pretability goals for detectors which leverage n-gram analysis and some of the most popular classification algorithms, namely; Logistic Regression, Random Forest, and Neural Networks. The stated interpretability goals were robustness, improving stakeholder confidence in the detector, and helping malware analysts with downstream tasks. A suggestion for future work is the specification of interpretation techniques for other machine learning algorithms or different feature sets.

In Chapter 5, a novel approach for providing fine a grain interpretation of malware detectors which leverage Convolutional Neural Networks was provided. The approach was able to out preform other similar methods in the literature while providing far better interpretations. The downside however was the technique was only usable on malware binaries within a certain size range. A technique which applies to binaries of any size is an objective for future research

In Chapter 6 we gave a novel approach for summarizing the robustness of a single binary classifier in a single metric. The binary classification case is common in the malware detection domain where models often classify samples as malicious or benign, however more work needs to be done on applying the metric to multi-class classification as well as regression tasks. This is necessary because as we have seen in Chapters 4 and 5, we are often interested in classifying malware by class as well.

Although machine learning has progressed greatly in the last decade or so, and despite the interest it generates for various high risk applications, including in cyber security, it is still a work in progress and there are many unsolved issues machine learning researchers face. The issue of interpretability is a complex one with no simple "one size fits all" solution. Even within the malware detection domain, interpretability solutions still differ largely from one model to the next. In this thesis we presented a step in the right direction but there is still much work to be done, in both making machine learning models interpretable and in making them practical for large scale cyber security applications. It is the authors hope that the discussion provided here is informative to the reader and helps them too form their own opinions about interpretability, inspires their own interpretability solutions, and in the end, helps machine learning based malware detectors play an essential role in cyber security.

VITA AUCTORIS

NAME:	William R. Briguglio
PLACE OF BIRTH:	Windsor, ON
YEAR OF BIRTH:	1995
EDUCATION:	Holy Names High School, Windsor, Ontario, 2014
	University of Windsor, B.Sc in Computer Science, Windsor, Ontario, 2019
	University of Windsor, M.Sc in Computer Science, Windsor, Ontario, 2020