Interpretable Machine Learning for Malware Detection and Adversarial Defense

Miles Qi Li, School of Computer Science

McGill University, Montreal

July, 2022

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of

Ph.D. of Computer Science

Abstract

As the Internet becomes ubiquitous and of paramount importance to people's lives, cyber attacks have also become a larger concern for not only individuals, but corporations and governments as well. As a consequence, cybersecurity has caught considerable attention. As machine learning has grown and large datasets have been amassed in recent years, an increasing number of advanced machine learning-based methods have been applied in the cybersecurity field. This proposed research focuses on proposing novel machine learning solutions with high performance for two cybersecurity tasks: malware detection and black-box adversarial defense. To help users of the prospective solutions gain insights into the cases and validate the outputs, the interpretability of the machine learning solutions is also evaluated in this research. All in all, this thesis makes new contributions in three directions: interpretable classification, malware detection, and black-box adversarial defense.

In recent years increasingly complex deep neural networks have been proposed to refresh the classification performance scoreboard in the field of applied machine learning. However, it is difficult to understand exactly how they make predictions. In some cases, interpretability is expected for multiple reasons, such as to gain trust in the classification results and to gain knowledge from the explanations. For interpretable classification, we propose an intrinsically interpretable feedforward neural network architecture that achieves both solid classification performance and interpretability.

Malware has been the major means for cyber attacks and there is tremendous growth in the volume of new malware broadcasting on the Internet. Thus, there is a pressing need

to create intelligent malware detection systems. Existing malware detection methods lack the ability to analyze malware based on their complete assembly code, and state-of-the-art methods also lack interpretability for classification results. To address these limitations, we propose a novel state-of-the-art deep neural network architecture that can model the full semantics of assembly code and that has the ability to analyze a sample from multiple static feature scopes as well as the interpretability to explain the detection results.

Machine learning models can be compromised by adversarial attacks that intend to cause them to mis-classify samples that contain often imperceptible but carefully selected perturbations. This vulnerability could be exploited to induce catastrophic consequences. Adversarial attacks can be conducted in either the while-box scenario, in which an adversary has complete knowledge of the target machine learning model, or the black-box scenario, in which an adversary has no knowledge of the target machine learning model. The latter is more common in real-world situations because most classification service providers do not reveal the details of their systems. Hence, our focus is on defense against black-box adversarial attacks. Existing defense methods are static and cannot dynamically evolve to adapt to adversarial attacks, which unnecessarily disadvantages them. In this segment of our research, we propose a novel dynamic defense method that can effectively utilize previous experience to identify black-box attacks.

Abrégé

Alors qu'Internet devient omniprésent capitale dans la vie des gens, les cyberattaques sont également devenues une préoccupation majeure non seulement pour les particuliers, mais aussi pour les entreprises et les gouvernements. Avec le développement de l'apprent -issage automatique et l'accumulation de grands ensembles de données ces dernières années, un nombre croissant de méthodes avancées basées sur l'apprentissage automatique ont été appliquées dans le domaine de la cybersécurité. Ce projet de recherche vise à proposer de nouvelles solutions d'apprentissage automatique pour deux problèmes de recherche en cybersécurité : la détection des logiciels malveillants et la défense adversariale en boîte noire. Pour aider les utilisateurs des solutions envisagées à mieux comprendre les cas et à valider les résultats, l'interprétabilité des solutions d'apprentissage automatique est également évaluée dans cette recherche. Cette thèse apporte de nouvelles contributions dans trois directions de l'apprentissage automatique pour la cybersécurité : la classification interprétable, la détection de logiciels malveillants et la défense contradictoire en boîte noire.

Ces dernières années, des réseaux neuronaux profonds de plus en plus complexes ont été proposés pour rafraîchir le tableau des performances de classification dans le domaine de l'apprentissage automatique appliqué. Cependant, il est difficile de comprendre exactement comment ils font des prédictions. Dans certains cas, l'interprétabilité est attendues pour de multiples raisons, telles que la confiance dans les résultats de classification et l'acquisition de connaissances à partir des les explications. Pour une classification interprétable, nous proposons une architecture de réseau neuronal à anticipation

intrinsèquement interprétable qui permet d'obtenir à la fois de bonnes performances de classification et une bonne interprétabilité.

Les logiciels malveillants sont à l'origine de nombreuses cyberattaques. Il est donc urgent de créer des systèmes intelligents de détection des logiciels malveillants. Les méthodes de détection de logiciels malveillants existantes n'ont pas la capacité d'analyser les logiciels malveillants sur la base de leur code d'assemblage complet, et les méthodes de pointe manquent également d'interprétabilité pour les résultats de classification. Pour remédier à ces limitations, nous proposons une nouvelle architecture de réseau neuronal profond à la pointe de la technologie qui peut modéliser la sémantique complète du code d'assemblage et qui a la capacité d'analyser un échantillon à partir de plusieurs portées de caractéristiques statiques ainsi que l'interprétabilité pour expliquer les résultats de la détection.

Les modèles d'apprentissage automatique peuvent être compromis par des attaques adverses qui visent à les amener à mal classer des échantillons contenant des perturbations souvent imperceptibles mais soigneusement sélectionnées. Cette vulnérabilité peut être exploitée pour entraîner des conséquences catastrophiques. Les attaques adverses peuvent être menées soit dans le cadre du scénario "while-box", dans lequel un adversaire a une connaissance complète du modèle d'apprentissage automatique cible, soit dans le cadre du scénario "black-box", dans lequel un adversaire n'a aucune connaissance du modèle d'apprentissage automatique cible. Ce dernier est plus courant dans les situations réelles car la plupart des fournisseurs de services de classification ne révèlent pas les détails de leurs systèmes. Par conséquent, nous nous concentrons sur la défense contre les attaques adverses en boîte noire. Les méthodes de défense existantes sont statiques et ne peuvent pas évoluer dynamiquement pour s'adapter aux attaques adverses, ce qui les désavantage inutilement. Dans ce segment de notre recherche, nous proposons une nouvelle méthode de défense dynamique qui peut utiliser efficacement l'expérience antérieure pour identifier les attaques de la boîte noire.

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Prof. Benjamin C. M. Fung, for accepting me to his research team and offering immense guidance and support to my Ph.D. study. I am deeply appreciative of the patience, efforts, and generosity that he has shown me since I came to Canada to begin my Ph.D. journey. I feel very lucky to learn from his extensive knowledge of the research area and benefit from his kindness towards his students. His support brings me serenity in this country.

I am also very grateful to my supervisory committee members, Dr. Xujie Si and Dr. Danny Tarlow, for their constructive feedback to my Ph.D. study. I would also like to thank the members of our DMaS lab, especially Steven Ding, Adel Abussita, Malik Altakrori, Ashita Diwan, Guillaume Breyton, and Alexander Blostein, for the insightful discussions on the research topics.

I would also like to express my thanks to my family. My parents, who are professors of natural science, told me that I had the talent to be a scientist and encouraged me to contribute to the advancement of science. They also told me that there was no need to be afraid of failure, and I could freely try my best to pursue a career as a scientist because they would always have my back, no matter what. Many thanks to my wife, who came to Canada to begin her Ph.D. study at McGill with me, for surrounding me with happiness and going through difficulties with me over the years. She has encouraged and inspired me from many aspects. Special thanks to my grandparents, Jiuyue Liu and Jinghua Yang, for their love since the beginning of my life.

I especially thank McGill University for accepting me as a member and for providing the high-quality education and opportunities to conduct research jointly with exemplary scholars.

This research is supported by Discovery Grants (RGPIN-2018-03872) from the Natural Sciences and Engineering Research Council of Canada, Canada Research Chairs Program (950-232791), and the Canadian National Defence Innovation for Defence Excellence and Security (IDEaS W7714-217794/001/SV1). The Titan Xp used for the research was donated by the NVIDIA Corporation.

List of Publications

During my Ph.D. study, I have completed seven papers as the first author. In these works, I am the one who proposed the ideas, implemented the methods, and conducted the evaluations. Among them, two journal articles, three conference papers, and one book chapter have been published/accepted. In addition, I am the second author of three other papers. Among them, one is published in a journal. All the relevant papers are listed below.

1. Interpretable Classification:

    - **Li, M. Q.**, Fung, B. C. M., & Abusitta, A. On the Effectiveness of Interpretable Feedforward Neural Network. In *Proceedings of the International Conference on Joint Conference on Neural Networks (IJCNN)*, 8 pages, Padova, Italy: IEEE, July 2022.
    - Abusitta, A., **Li, M. Q.**, & Fung, B. C. M. Survey on Explainable AI: Techniques, Challenges and Open Issues. (**Under review**)

2. Malware Detection:

    - **Li, M. Q.**, Fung, B. C. M., Charland, P., & Ding, S. H. H. I-MAD: Interpretable Malware Detector Using Galaxy Transformer. *Computers & Security (COSE)*, 108(102371):1-15, September 2021. Elsevier. (JCR impact factor: 4.438)
    - Abusitta, A., **Li, M. Q.**, & Fung, B. C. M. Malware Classification and Composition Analysis: A Survey of Recent Developments. *Journal of Information Security*

and Applications (JISA), 59(102828):1-17, June 2021. Elsevier. (JCR impact factor: 3.872)

3. Black-box Adversarial Defense:

    - **Li, M. Q.**, Fung, B. C. M., & Charland, P. DyAdvDefender: An Instance-based Online Machine Learning Model for Perturbation-trial-based Black-box Adversarial Defense. *Information Sciences (INS)*, 601:357-373, July 2022. Elsevier. (JCR impact factor: 6.795)

The following research articles were also completed during my Ph.D. study, but their contents are not included in the thesis.

1. Malware Classification:

    - **Li, M. Q.**, Fung, B. C. M., Charland, P., & Ding, S. H. H. A Novel and Dedicated Machine Learning Model for Malware Classification. In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT)*, pages 617-628, Virtual Conference: ScitePress, July 2021. (Full paper acceptance ratio: 15%)
    - **Li, M. Q.**, Fung, B. C. M. *Software Technologies*, chapter: A Novel Neural Network-based Malware Severity Classification System, pages 218-232, series Communications in Computer and Information Science (CCIS), July 2022. Springer.
    - **Li, M. Q.**, Fung, B. C. M. Interpretable Malware Classification based on Functional Analysis. In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT)*, 8 pages, Lisbon, Portugal: July 2022.

2. Vulnerability Detection:

    - **Li, M. Q.**, Fung, B. C. M., & Diwan, A. A Novel Deep Multi-head Attentive Vulnerable Line Detector. (**Under review**)
    - Diwan, A., **Li, M. Q.**, & Fung, B. C. M. VDGraph2Vec: Vulnerability Detection in Assembly Code using Message Passing Neural Networks. (**Under review**)

Contribution to Original Knowledge

The contents in this thesis contribute to the original knowledge base of three research communities: interpretable machine learning, malware detection, and adversarial attack and defense.

For interpretable classification, the proposed *interpretable feedforward neural network* (*IFFNN*) architecture can serve as a novel solution for many classification scenarios that require both high classification performance and explanations for the classification results. It also has two advantages over the most prevalent post-hoc explanation methods. One is that the explanations are genuine and thus accurate. The second is that there is very little extra computation required to obtain the explanations. Our comprehensive evaluation on the classification performance and interpretability of the proposed solution also shows the effectiveness of our proposed approach to the research community.

For malware detection the proposed overall solution, namely *I-MAD*, includes several novelties and can serve as a new state-of-the-art malware detector for practical use. Its *Galaxy Transformer* component stands as an early effective attempt to model the full sequences of assembly code for malware detection. It also includes our improved ways to use traditional features for malware detection. Its interpretability allows malware analysts to examine the classification results and gain insights into the malware samples.

For black-box adversarial defense, we propose a novel state-of-the-art defense method, namely *DyAdvDefender*, an instance-based online machine learning model. As compared to previous defense methods, it is the first instance-based online machine learning model for the task. Its advantage is the ability to update its state whenever it receives a new

query sample. It outperforms previous defense methods against different adversarial attack algorithms, and it can serve as a practical solution for black-box adversarial defense to be applied in real-world applications.

Contribution of Authors

The novel solutions to the three research topics involved in the thesis were proposed by the candidate after discussions with my supervisor Prof. Benjamin C. M. Fung. The candidate then implemented the proposed solutions, conducted the experiments, improved the initial solutions, and formed the research papers that have been published/accepted in a journal/conference proceedings under the supervision of Prof. Fung. The candidate then wrote the corresponding chapters in the thesis based on the published/accepted papers. Dr. Adel Abusitta, a postdoctoral fellow in our lab at McGill, provided validation and suggestions on improving the writing of the research paper for interpretable classification. Mr. Philippe Charland, a defense scientist from Defence Research and Development Canada (DRDC), provided feedback to the candidate on the research projects of malware detection and black-box adversarial defense, and he helped the candidate improve the writing of the corresponding research papers. Dr. Steven H. H. Ding helped the candidate on the acquisition of the datasets used in the malware detection work and provided suggestions on the improvement of the writing of the research paper.

The literature reviews of the three research topics in the thesis are all written by the candidate under the supervision of Prof. Benjamin Fung. The literature review on malware detection was written to fulfill the comprehensive exam required by the School of Computer Science. Dr. Adel Abusitta then identified the values for the community and improved it jointly with the candidate to form a survey paper published in a journal. Dr. Adel Abusitta and the candidate also completed a survey paper on explainable AI. The literature review on interpretable and explainable machine learning in the thesis is dis-

tinct from this survey paper and drafted by the candidate himself. The literature review on adversarial attack and defense has been written by the candidate himself as well.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

IFFNN	Interpretable Feedforward Neural Network
Malware	Malicious Software
I-MAD	Interpretable MAlware Detector
DyAdvDefender	Dynamic Adversarial Defender
GA^2Ms	Generalized Additive Models with Pairwise Interactions
CNN	Convolutional Neural Network
RETAIN	REverse Time AttentIoN
RNN	Recurrent Neural Network
NBC	Naive Bayes Classifier
DT	Decision Tree
FGSM	Fast Gradient Sign Method
IGS	Iterative Gradient Sign
PGD	Projected Gradient Descent
FDM	Forward Derivative Method
BN	Bayesian Network
KNN	K-nearest Neighbors

SVM	Support Vector Machine
PE	Portable Executable
VM	Virtual Machine
TF-IDF	Term Frequency-Inverse Document Frequency
COFF	Common Object File Format
DeepLIFT	Deep Learning Important FeaTures
ZOO	Zeroth Order Optimization
AutoZOOM	Autoencoder-based Zeroth Order Optimization Method
NES	Natural Evolution Strategies
SAP	Stochastic Activation Pruning
CGAN	Conditional Generative Adversarial Network
KD	Kernel Density
CAM	Class Activation Mapping
LIME	Local Interpretable Model-agnostic Explanations
SHAP	SHapley Additive exPlanations
ERM	Empirical risk minimization
LSH	Locality Sensitive Hashing

Chapter 1

Introduction

The growth and success of the Internet has brought enormous benefits and convenience to modern life. People connect computers, smartphones, and tablets to the Internet for various purposes such as communications, information, services, businesses, and more. According to the statistics, the average person spends around 7 hours looking at a screen per day [99]. The Internet has penetrated into every aspect of daily life, and this fact raises security concerns. Due to the ubiquitous Internet, malicious hackers have the opportunities to cause destructive losses to individuals, businesses, and governments as well. In fact, cyber attacks are undeniably happening on a daily basis [1].

On the other hand, the booming success of the Internet also allows data scientists to gather large volumes of data for analyses. This yields the opportunity for the development of state-of-the-art machine learning techniques to create novel defense systems against cyber attacks [53, 96]. The objective of this thesis is to propose novel machine learning solutions to two major cybersecurity tasks: malware detection and black-box adversarial defense. As a classification task, existing state-of-the-art malware detection solutions do not have the interpretability to explain the classification results. It is widely believed that there is a trade-off between performance and interpretability of machine learning models [119, 128]. In the applied machine learning community, interpretabil-

[1] https://www.fireeye.com/cyber-map/threat-map.html

ity is often compromised for better performance. However, from our collaboration with malware analysts, we see that analysts expect the explanations of classification results to validate the results and gain insights into the target samples. Therefore, we value the interpretability of the solutions in this thesis. To overcome the challenge of keeping both excellent performance and interpretability we propose a novel neural network architecture for interpretable classification to be the foundation for the classification task, and it is thus applied to our solution for malware detection.

Below, we introduce the three tasks: interpretable classification, malware detection, and black-box adversarial defense and our solutions to them.

1.1 Interpretable Classification

Deep learning classification models are achieving state-of-the-art performance in an increasing number of tasks [21, 56, 75]. They usually work as black-boxes; when a large number of training samples are fed to them, they learn patterns that correlate with different classes. The patterns are then used to classify unseen samples. However, most deep neural networks only implicitly learn and use the patterns, and they do not explicitly explain the reasons why a sample belongs to a class.

Having said that, there are interpretable machine learning classification models, such as linear regression, softmax regression, and decision trees [84]. These models can explain their classification results in a clear and simple way. However, their expressive abilities are very limited, so they cannot model complex relations between a feature and the outcome or the various interactions between different features. For example, linear regression and softmax regression can be seen as neural networks with no hidden layers. They can tell to what extent each feature contributes to a classification result. The interpretability comes from the fact that the relation between a feature and the class of a sample is computed independently without any interactions. Even though this simplicity allows the models to explain their classification results, they yield inferior results as

compared to multi-layer neural networks. In this day, classification performance has a higher priority than interpretability in most cases. Hence, these simple models are often less useful than complex and non-interpretable models [98].

In an attempt to solve the dilemma of choosing between high classification performance or interpretability, some post-hoc explanation techniques have been proposed to explain the classification results of complex machine learning models, but these methods have several limitations. Some of them have high computational complexity, and their derived explanations are not always accurate [44, 120, 133]. Some of them can only explain certain kinds of models [16, 149].

To avoid the aforementioned methods, we propose a novel intrinsically interpretable feedforward neural network architecture to provide genuine explanations for the classification results without harming the classification performance and with little overhead to acquire the explanations. We conducted comprehensive experiments to illustrate that the explanations are accurate and that the neural network architecture required for the interpretability does not harm the classification performance.

1.2 Malware Detection

Malicious software (malware) is any software that is designed to cause damages to a computer system or its users. Specifically, malware can block an Internet connection, corrupt an operating system, steal a user's private information, and encrypt a user's important documents for ransom. In the last 20 years, malware has been a growing threat to computer users, and in 2017 the number of new malware increased by 22.9% over 2016 to 8,400,058 [2]. Recognizing malware samples downloaded by legitimate users in a timely manner is of crucial importance for users' protection. To manually analyze malware samples is not efficient enough to prevent such a large number of new malware samples from releasing their payloads and causing damages. There is thus a pressing need to create artificial intelligence-based methods to recognize malware.

Signature-based malware detection methods have been widely used in antivirus products. They can recognize known malware, but they are limited in recognizing significant variants of existing malware and new malware [46, 146]. Therefore, machine learning-based methods have been proposed.

There are two major challenges these methods face. The first is to model the full semantics behind the assembly code of malware. This is because the whole assembly code of executables is very long, and an executable of 1 MB could contain hundreds of thousands of assembly instructions. The second challenge is to provide interpretable results while keeping excellent classification performance. This is because in most cases there is a trade-off between the classification performance of a machine learning model and its interpretability [10, 119, 128]. Interpretability is one of the dominant features for classification models in some domains, such as healthcare and cybersecurity. In malware detection, the interpretability can help malware analysts examine the classification results and create a knowledge base of malware samples.

We propose the following novel solutions to overcome these challenges and provide better performance:

- a novel neural network architecture called the *Galaxy Transformer* network that uses the innate hierarchical structure of the assembly code of a sample to compute its vector representation;
- improved ways to use traditional features for malware detection;
- application of our proposed interpretable feedforward neural network to provide explanations on the importance of each feature to a detection result.

Experiment results show that our overall model, *I-MAD*, significantly outperforms existing state-of-the-art static malware detection models and presents meaningful explanations.

1.3 Adversarial Attack and Defense

Machine learning models, especially deep neural networks, have achieved state-of-the-art performance in many fields in the cyber world, e.g., image classification, malware detection, and natural language processing [22, 88, 145]. As a consequence, state-of-the-art machine learning models have gradually been integrated into online services. As this transformation brings convenience to people's lives, it also comes with a concern. Researchers have found that machine learning models can be compromised by adversarial attacks [61, 82, 109, 152] that intend to cause the machine learning models to misclassify samples that contain often imperceptible, but carefully selected perturbations. This vulnerability can be exploited to induce catastrophic effect: an autonomous vehicle that fails to recognize a stop sign may cause a traffic accident; failure to recognize a malware program may cause an enormous loss to a computer user. Evidence proves that adversarial attacks have come to real-world scenarios [82, 134]. As a result, adversarial attack and defense are topics extensively studied [142].

A machine learning classification model is a function $f(\boldsymbol{x})$ that maps an input sample $\boldsymbol{x}$ to an output $\boldsymbol{y}$, an n-dimensional vector, where n is the number of classes. Each element of $\boldsymbol{y}$ corresponds to the probability that $\boldsymbol{x}$ belongs to a certain class. The objective of an adversary is to add a minimal perturbation $\boldsymbol{\delta}$ to a natural sample $\boldsymbol{x}$, so that $f(\boldsymbol{x} + \boldsymbol{\delta})$ presents a false result. Adversarial samples with large perturbations are relatively easy to detect; however, adversarial attacks present a challenge because the perturbation added to a natural sample is usually very small, yet very effective [28, 142].

A common issue with existing defense methods is that they are static; therefore, they cannot update their states to counter adversaries. This leads to an unnecessary disadvantage for the defenders because adversaries can update their attack strategies according to the information acquired from their pry attempts. Based on this intuition, we propose *DyAdvDefender*, the first instance-based online machine learning model for the defense against black-box adversarial attacks. DyAdvDefender can recognize a perturbed sam-

ple that originates from the same sample as a previously queried sample so as to prevent the adversaries from prying gradient information. Extensive experimental results suggest that DyAdvDefender outperforms existing static methods in terms of defense effectiveness while keeping the original classification accuracy with only limited extra time consumption.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2 presents a literature review on the research topics relevant to this thesis.

Chapter 3 presents the proposed interpretable feedforward neural network (*IFFNN*) architecture for interpretable classification. Part of the contents in this chapter has been accepted to the 2022 International Joint Conference on Neural Networks (*IJCNN 2022*) as a full paper.

Chapter 4 presents the details of our *Interpretable MAlware Detector* (*I-MAD*) and demonstrates how it addresses the challenges for malware detection. Part of the contents in this chapter has been published in *Computers & Security*, an Elsevier journal.

Chapter 5 presents *DyAdvDefender*, our instance-based online machine learning model to defend against black-box adversarial attacks, and demonstrates how it is optimized for efficiency and effectiveness. Part of the contents in this chapter has been published in *Information Sciences*, an Elsevier journal.

Chapter 6 concludes the proposed solutions for the research topics in this thesis and discusses the potential improvements that could be made as future work.

Chapter 2

Literature Review

In this chapter, we present a literature review on each of the three research topics concerning this thesis. Section 2.1 reviews existing interpretable and explainable machine learning techniques. Section 2.2 reviews the features and machine learning models that have been proposed for malware detection and classification. Section 2.3 reviews the existing adversarial attack and defense methods in different scenarios.

2.1 Interpretable and Explainable Machine Learning

In the literature of applied machine learning, most works are aimed at state-of-the-art performance. The gradually deeper and more sophisticated deep neural networks have dominated the machine learning community for 10 years [21, 56, 75, 151]. They usually work as black-boxes, with the ability to learn patterns that correlate with different classes from a large number of training samples, and use them to classify unseen samples. However, most deep neural networks only implicitly learn and use the patterns, and cannot explain the rationale behind a prediction.

There are situations in which explanations for the predictions of machine learning models are required. For example, in healthcare the predictions of machine learning models concern critical decisions for patients, and thus they need to be justified by a doctor.

For malware detection, explanations of the detection results can help malware analysts examine the detection results and gain insights into malware samples.

Explanations of machine learning models can be categorized as local and global [98]. Local explanations provide information on how an individual prediction is made, while global explanations describe how a machine learning model makes predictions in general. For the aforementioned purposes, we focus on local explanations in this thesis.

Many researchers believe that there is a clear trade-off between performance and interpretability of machine learning models [90, 128], while some others do not agree [122]. As a matter of fact, linear or piece-wise linear models are intrinsically interpretable, and the models that achieve state-of-the-art results are not intrinsically interpretable. To understand the rationale behind the predictions of the complex black-box models, post-hoc explanation techniques can be used. They form the field of explainable machine learning, and the intrinsically interpretable models form the field of interpretable machine learning.

In the remainder of this section we review the literature on the techniques to acquire local explanations for machine learning models. Following the terminology used by Rudin [122], we refer to the machine learning models that can provide genuine explanations by themselves as "interpretable machine learning", and refer to the post-hoc techniques to explain other machine learning models as "explainable machine learning".

2.1.1 Interpretable Machine Learning

Linear regression, logistic regression, and softmax regression are linear models for regression, binary classification, and multi-class classification, respectively. They are completely interpretable because the product of the weight of a feature and the value of the feature directly shows the contribution of the feature to the prediction. However, the critical problems with these linear models are that they assume 1) the relation between a feature and the prediction is linear, and 2) the features are independent of each other towards determining the prediction. These assumptions are not true in most real-world

applications. Thus, the expressive ability of the linear models limits their performance in real-world applications.

Generalized additive models with pairwise interactions (GA^2Ms) can address the limitations of linear models to some extent [25]. GA^2Ms map each feature x_i to a potentially higher order space with a function $f_i(x_i)$, and thus establish non-linear relations between the features and the prediction. To allow feature interactions, the features are paired to form new features $f_{ij}(x_i, x_j)$ to be used the same way as the original features. Caruana et al. [25] use these ideas together with other techniques to achieve state-of-the-art performance on hospital 30-day readmission prediction. That being said, the limitation of their method is that the interaction between more than two features is not modelled. If the interactions of any feature combination are similarly included, the complexity would make it unacceptable. In this sense, feedforward neural networks have the advantage because they can model any kind of interaction between the features [40].

Another type of interpretable model with better expressive ability than linear models is decision trees [92]. Decision trees split the values of features into different intervals, and based on the intervals in which the features of a sample locate, the class of the sample is determined. The way a decision tree makes a prediction can be expressed as a set of "if-then" rules that are easily understandable to human experts. The decision boundary of a decision tree is thus composed of straight lines. This kind of boundary allows the model to form a better approximation to real-world data distribution than linear models. However, the practice of splitting continuous variables into intervals causes the loss of information, and thus inevitably reduces the modelling accuracy.

Zhou et al. [150] propose a way to make *convolutional neural networks* (*CNNs*) interpretable. It requires the top layers of a CNN to have a constrained form: the feature maps of the penultimate layer should be followed by a global average pooling, and the output is fed to a fully-connected layer that produces the logits for classification. The weighted summation of the feature maps for each class is called the class activation map

(CAM). The CAM is upsampled to the shape of the input image to show how discriminative/important each region of the input image is for the prediction.

For the interpretable classification of sequential data, Choi et al. [34] propose the *REverse Time AttentIoN* (*RETAIN*) model that is composed of two attention-based *recurrent neural networks* (*RNNs*) to form a softmax regression with "dynamically" computed weights. One attention-based RNN is used to compute the importance/contribution of each item in the sequence to the classification result, and the other attention-based RNN is used to compute the importance/contribution of each variable in an item of the sequence to the item. Then, they are combined to dynamically form the weights of the variables in the items to be used in a softmax regression for the final classification result. The proposed method delicately achieves the objective of a non-linear mapping from the features to the prediction and has the interpretability to explain the prediction. The problem with this work is that the explanations are not evaluated, and thus their quality is unclear. This work also motives us to propose our novel solution for the interpretable classification of vectorial data and to conduct a comprehensive evaluation on the validity of its interpretability.

2.1.2 Explainable Machine Learning

In the literature of explainable machine learning, post-hoc explanation methods are categorized as model-agnostic and model-specific methods depending on whether the explanation method can be applied to any machine learning model for a task or only to a specific family of machine learning models.

Model-Agnostic Explanation Methods

Model-agnostic explanation methods treat the target machine learning model as a black-box, i.e., they do not require the knowledge of the model to explain, and only need to have the output of the model given any valid input to estimate how the target model

makes a prediction. The common disadvantages of this type of explanation method are as follows:

- The fidelity of the explanations is not guaranteed.
- Most of them need to run the target model many times to acquire an explanation.

Below are some representative model-agnostic explanation methods.

Ribeiro et al. [120] propose a model-agnostic explanation method called *Local Interpretable Model-agnostic Explanations* (*LIME*). To explain the prediction of a sample, they try to randomly generate samples around the neighborhood of the target sample in the feature space. They first use the target model to label these samples, and they then train an intrinsically interpretable model with these samples and the labels given by the target model. The intrinsically interpretable model is considered to be a local surrogate of the target model. The explanation given by the surrogate model is used to explain the target model. The obvious drawback is the high time complexity of the method, especially when the feature space is huge.

To observe how the value of a feature marginally affects the prediction for a sample, Goldstein et al. [60] propose to plot a curve of the prediction against the feature while keeping other feature values static. By applying the method on all features one by one, they can establish an idea of how each feature influences the prediction. This method is straightforward; however, it ignores the potential correlation between the feature to observe and other features. The conclusion from the plot can be very misleading when there is strong correlation between the features.

Lundberg and Lee [94] propose SHAP (SHapley Additive exPlanations) values to explain the contribution of each feature to the prediction of a sample. Shapley value is a concept in cooperative game theory. It is the average marginal contribution of a feature value across all possible coalitions. SHAP values are the Shapley values of a conditional expectation of the target model. In other words, SHAP values are the marginal contribution of a feature value to the prediction of a target sample when other features have the exact

values in the target sample. The authors propose two model-agnostic and four model-specific methods to estimate SHAP values. SHAP values have three desirable properties to serve as contributions of features. However, this method has very high computational complexity, and thus it is not entirely practical.

Model-Specific Explanation Methods

The application scopes of different model-specific explanation methods can be quite different. For example, some methods are proposed to explain CNNs [128, 131, 149], while some others can explain all neural networks (i.e., differentiable models) [130, 133]. Even though they are all considered to be model-specific, we believe that the latter should be categorized as model-semi-specific or model-semi-agnostic. Sometimes it can be difficult to determine if an explanation method is model-specific or if the target model is intrinsically interpretable because in both cases the explanation technique can only be applied to explain a specific model. Intuitively, intrinsically interpretable models are somehow constrained in their forms, and the explanations given to an intrinsically interpretable model are genuine. That is why CAM and RETAIN, which we introduced above, are categorized as intrinsic interpretability.

Sundararajan et al. [133] propose integrated gradients as a method to explain any type of neural networks. They aggregate the gradients along the input space that fall on the straight line between a baseline sample (i.e., a sample of which the prediction given by the target model is neutral) and the target sample to serve as the contribution of the features to the prediction. They prove that integrated gradients is the axiomatic explanation of the prediction made by neural networks.

Shrikumar et al. [130] propose another method to explain any neural network, called *DeepLIFT* (*Deep Learning Important FeaTures*). It can assign a contribution value for each neuron or input feature for the prediction. This is achieved by comparing the activation of each neuron of a target sample with the activation of a neutral reference sample, and it

then calculates a contribution score according to the difference. The contribution values can be backpropagated with chain rule until the input features.

Deconvnet [149] and guided backpropagation [131] are proposed to explain convolutional neural networks. For each kind of layer in a convolutional neural network, an inverse operation is proposed to compute the contribution of a neuron in the lower layer for the activation of neurons in the higher layer. The inverse computation goes downwards until the input feature map to show the contribution of the input features.

Selvaraju et al. [128] propose *Grad-CAM* (*Gradient-weighted Class Activation Mapping*), a generalization of CAM [150], to explain the prediction of general CNNs by showing the important regions of an image for the prediction. They perform global average pooling on the gradient of a target class with respect to a convolutional feature map to serve as the importance weight of the feature map for the target class. Then, the weighted summation of the feature maps at the same layer are used to indicate the important regions to the target class. As compared to CAM, Grad-CAM can be applied to general CNNs without the constraints of form for CAM.

To make *support vector machines (SVM)* interpretable, Chen and Wang [33] identify that a classifier based on fuzzy rules implicitly defines a Mercer kernel that is translation invariant as long as the classifier's membership functions satisfy an assumption. Thus, they can build an interpretable SVM based on the learned fuzzy rules. The resulting model has the expressive ability of a non-linear SVM and can provide explanations with the fuzzy rules.

2.1.3 Interpretable Machine Learning vs. Explainable Machine Learning

Both interpretable machine learning and explainable machine learning have advantages and disadvantages as compared with each other. Model-agnostic post-hoc explanation methods can be applied to explain any machine learning model, thus giving flexibility to

the choice of the machine learning model for prediction. However, the fidelity of the explanations that they provide may not be accurate [122], and they usually require running the target model many times to acquire an explanation, which is quite time-consuming. Model-specific explanation methods may also have these issues to a lesser degree. Intrinsically interpretable models provide genuine, and thus accurate explanations; however, they are constrained in their form to keep their interpretability, and hence some of them have lower prediction performance.

2.2 Malware Detection

Most antivirus engines detect or classify malware by checking whether the files to be analyzed contain malware signatures. Those signatures are created by human antivirus experts (known as malware analysts) through carefully examining the collected malware samples. The signatures of malware can be filename, text strings, regular expressions of byte sequence, etc [35,127]. Obviously, signature-based methods can only detect malware originates from known malware which does not change significantly. However, malware can hide its malicious behavior through obfuscation, packing, polymorphism, metamorphism, etc., so that the malware variants look entirely different from their original versions. Therefore, signature-based methods have two shortcomings. First, they have high precision but low recall rate. Second, the process of creating signatures is labor-intensive. Considering there is a large number of new malware samples appear every day, there is a pressing need to develop new intelligent malware analysis methods.

To alleviate the burden of manual signature crafting, researchers propose different automatic signature generation methods [27,30,147]. The content of the signatures can be windows system call combinations [147], control flow graphs [27], and functions [30].

Researchers also propose to use machine learning models to detect and classify malware [7,41,42,69,72,77,78,101,106,125–127]. Generally speaking, machine learning models have more flexibility than signature-based method since a signature-based method

usually examine whether any signature in the database exist in an unknown sample to determine whether it is positive while machine learning models can examine multiple feature sources (e.g., printable strings, assembly code, windows system calls, etc.) and they synthesize the effects of all features for the classification. As deep learning models gradually outperform other machine learning models in most fields in the last 10 years, effective deep learning models for malware analysis have been explored [41,69,77,126].

The scope of the literature review consists of publications on malware detection and classification techniques. Malware detection refers to the binary classification task to determine whether an unknown executable is malware while malware classification refers the multiclass classification task to classify an executable to one of the pre-defined classes which may include specific malware classes (e.g., families), a generic malware class, and a benign software (benignware) class. The techniques for both tasks are interchangeable since the effective features for one of them are also effective for the other and the models for both of them are classification models, so we do not distinguish them too much in this literature review. We include publications that contribute on new feature extraction methods, new feature sets (signatures are included when we use the term 'features' in the paper), and new models for malware detection and classification.

The literature review is organized as follows. In Section 2.2.1, we review some common techniques used in malware, and with that knowledge we can better analyze each component of a malware analysis system. In Section 2.2.2, we compare the static and dynamic feature extraction methods. In Section 2.2.3, we review the features used in malware analysis systems and their representations. In Section 2.2.4, we review the classification models used for malware analysis.

2.2.1 Malware Techniques

In this section, we introduce the common techniques used by malware authors to evade the detection.

Obfuscation

The term 'obfuscation' in this literature review refers to the techniques used to create a variant of the original code without affecting its functionality and it does not include other techniques, such as packing, polymorphism, and metamorphism. The purpose of obfuscation is usually to hide the real logic of the original code or to evade signature-based detector or function clone detector [29, 52]. A few commonly used obfuscation techniques are as follows:

1. Dead-Code Insertion [35]: insert useless instructions (e.g., nop) or insert some instructions that only affect unused variables.

2. Code Transposition [35]: change the order of the independent instructions.

3. Register Reassignment [35]: exchange the usage of registers for the storage of data/address in a specific live range.

4. Instruction Substitution [35]: replace an instruction with equivalent instructions.

5. Control Flow Flattening [83]: 1) break up the body of the function to basic blocks 2) put all basic blocks which were originally at different nesting levels, next to each other 3) encapsulate all basic blocks in a selective structure (a switch statement in the C++) 4) encapsulate the selection in a loop

6. Bogus Control Flow [1]: for a basic block, add a new basic block which contains an opaque predicate and then makes a conditional jump to the original basic block.

Packing

Packing is a technique to compress or encrypt an executable and put an uncompressing or decryption engine in it so that those packed files will be uncompressed/decrypted during runtime. It means a static analyzer cannot see the real program since a static analyzer does not run the executable. Packing is used not only for malware but also for the protection

of benignware schemes [8, 27]. According to the statistics of Anderson et al. [8], 47.56% malware is packed and 19.59% benignware is packed.

Polymorphism

Polymorphism is also a technique based on encryption and decryption. A polymorphic malware contains two parts: the polymorphism engine and the real program which performs the malicious functions. The former mutates the encryption algorithms and keys when it replicates and the code of the latter per se is fixed but it is encrypted by the former in different ways in each generation, so the same polymorphic malware programs of different generations appear completely different [89].

Metamorphism

A metamorphic malware re-programs itself when it replicates, so in each generation, the whole program body is modified using code obfuscation techniques while the functionality is kept [89]. Metamorphic malware is considered to be more difficult to write than polymorphic malware and polymorphic malware is easier to be detected since the main program of polymorphic malware keeps the same after it is dynamically decrypted.

2.2.2 Feature Extraction Methods

Features of executables can be extracted statically or dynamically. In this section, we compare the advantages and disadvantages of those two feature extraction methods.

Static Extraction Methods

A static feature extraction method extracts features from the content of an executable without running it. Features can be extracted utilizing the file format, e.g., *Portable Executable (PE)*, *Common Object File Format (COFF)* so that code, data, information from metadata including DLLs, DLL functions, etc., can be extracted accordingly [8,72,101,126,127].

IDA Pro[1] is a commonly used disassembler to extract assembly code, data, or printable strings [8, 72, 101]. The static features can also be extracted in a file-format agnostic manner. Features extracted this way can be byte sequences, file size, byte entropy, etc [78,106,126,127]. The advantage of static feature extraction method is that it can cover the complete content of a file. However, the problem is static extraction methods are prone to packing and polymorphism since most features statically extracted from a packed or polymorphic malware sample are from encrypted contents rather than the original program body [18].

Dynamic Extraction Methods

A dynamic feature extraction runs an executable usually in an insulated environment which can be a virtual machine or an emulator and then extract features from the memory image of the executable, the executed instructions, the high-level behaviors, and/or the changes of system state. Since malware equipped with packing or polymorphism has to exhibit the real malicious program to achieve their goals, dynamic feature extraction is more resistant to those malware techniques compared with static feature extraction method [18].

A virtual machine (VM) provides a subset of real hardware resources of a computer in an insulated environment, so VMs have the limitation that they do not support cross-platform analysis since the guest and host CPUs must be the same [27]. Anderson et al. [7, 8] use Xen[2] and Royal et al. [121], Dai et al. [42], and Islam et al. [72] use VMWare[3] to create and run their VMs to perform dynamic analysis. Kolosnjaji et al. [77] extract system call sequences using Cuckoo sandbox[4] which is an open source automated malware analysis system. Other researchers who work for a commercial anti-virus engine use the VMs as part of their anti-virus engines to dynamically extract features [41,69].

[1]https://www.hex-rays.com/products/ida/
[2]https://www.xenproject.org/
[3]https://www.vmware.com/
[4]https://cuckoosandbox.org/

An emulator uses software to perform functions of hardware. There are two categories of emulators: full-system emulator and application level emulator. A full-system emulator is a piece of software that emulates every component of a computer, including its processor, memory, graphics card, hard disk, etc., with the purpose of running an unmodified operating system. Qemu[5] is a full-system emulator used by several malware analysis systems [18, 57, 125]. Considering the time-consuming of full-system emulator, Cesare and Xiang [27] propose to use application level emulation to unpack malware more efficiently so that only the parts which are necessary to execute the file including the instruction set, virtual memory, windows API, linking and loading, thread and process management, and OS specific structures are implemented.

One shortcoming of dynamic feature extraction methods is that it does not reveal all possible execution paths [18]. Malware may have detection routines to check whether it is executed in a VM or an emulator. When malware finds out that it is executed in such an environment, it may halt its execution so dynamic models will fail to recognize it as malware. The methods to detect whether an executable is executed inside a VM can be found from several papers [55,124]. According to Bayer et al. [18], it is harder for malware to find it is executed in an emulator than in a VM. In addition, some malware behavior is triggered only under certain conditions and this may fail to be captured in an insulated environment.

Another problem of the dynamic methods is that they take much more time than static feature extraction [18]. Specifically, analysis with emulators takes more time than with VMs.

2.2.3 Features

In this section, we introduce the features used by malware analysis systems and how they are extracted and represented.

[5]https://www.qemu.org/

Printable Strings

A printable string is a sequence of ASCII characters terminated with a null character. Schultz et al. [127] find that malware has similar strings that distinguished it from benignware and the latter also has common strings that distinguishes it from malware. Printable strings are represented as binary features in which '1' represents a string is present and '0' represents it is absent in all work [41,69,72,127] except the work of Saxe and Berlin [126].

Schultz et al. [127] extract printable strings from the headers of PE files. The extraction is executed straightforwardly since the header is in plain text format. Dahl et al. [41] and Huang and Stokes [69] extract null-terminated objects from the memory image of a malware process [41,69] as printable strings. The coverage of their methods is better than extract printable strings just from the headers but there are some false positive results extracted this way. Huang and Stokes [69] find that most null-terminated objects are indeed unpacked strings, but a few are individual code fragments. Dahl et al. [41] find that most null-terminated objects correspond to system strings used to create a file. Islam et al. [72] use the strings utility in IDA Pro to extract printable strings from a whole file.

Different from other works, Saxe and Berlin [126] do not take printable strings as binary features but use their hash values and the logarithm of the string lengths to create a histogram. The hash values are integers from 0 to 15. For the length axis, they start the histogram bin bounds at length 8 and evenly space the bins between 8 and 200, except that the last bin's upper bound extends to infinity. They use the value of each bin of the histogram as a feature. They take all byte sequences of length 6 or more that are in the ASCII range as printable strings which is also different from other works. The rationale behind this kind of feature representation is questionable since entirely different printable strings are mapped to the same bin so the value of each bin is the count of a mixture of irrelevant printable strings.

Essentially, the functionality of most malware does not rely on fixed printable strings. When malware creators find that some strings are used by malware detectors, they can

change them or encrypt them. Therefore, printable string features alone are not sufficient for malware detection.

Byte Sequences

Executables consist of byte sequences. A byte sequence may belong to the metadata, code sections, data sections, etc. As has been stated, byte sequences are important signatures of malware since malware may share some common sequences that are the same or follow the same regular expression. Likewise, byte sequences are also informative to be features for malware analysis systems [8,78,126,127].

Schultz et al. [127] use bigram byte sequences in the form of binary features and they claim byte sequences are the most informative because it represents the machine code in an executable. Strictly speaking, the reason they give is not entirely correct since some byte sequences come from metadata, data sections, etc. Even if a byte sequence is from code section, many byte sequences do not align with a machine code instruction since the instructions have variable lengths in some architectures. Moreover, their byte sequence features have the problem of dimension explosion since there are too many different bigram byte sequences and it is too large to fit them into memory when they publish their work so they could only split the byte sequence set into several subsets and feed them to multiple naive Bayes models.

To solve the dimension explosion problem, Kolter and Maloof [78] use information gain to select the top 500 informative 4-gram byte sequences as features from 256 million distinct 4-grams. They use them as binary features for all models they tried and also use term *frequency-inverse document frequency (TF-IDF)* of those 4-gram byte sequences as an additional feature representation for their KNN classifier.

Different from the above two works, Anderson et al. [8] do not use byte sequences per se as features but fit byte sequences into a Markov chain so they get the transition probabilities from one byte to another. The transition probability matrix is used as a kind of feature.

Saxe and Berlin [126] slide a 1024 byte window over an input binary, with a step size of 256 bytes. Then they compute the byte entropy of each 1024 byte window and the occurrence of each byte to form a histogram and evenly separate each axis into 16 bins to form a 256 length feature vector.

Nataraj et al. [106] make the whole byte sequence of a file into a picture in which each byte represents the grey scale of a pixel. They find malware belonging to the same family appear very similar in layout and texture. The width of the image used to transform the 1D byte sequence into a 2D matrix (image) is determined by the size of the file. The texture feature of the malware image is computed using the algorithm proposed by Oliva and Torralba [107]. One fatal drawback of this method is that the image of a file is not stable, e.g., if there is a manual offset in any section of the file or even one insertion or deletion of one instruction, the texture can be entirely distorted.

Byte sequences are not reliable features either. Obfuscation techniques such as instruction substitution and register reassignment can change the opcodes and operands respectively and thus the machine code is entirely changed. In all those works, the byte sequences are statically extracted but the main program body encrypted with different algorithms or keys through Packing and Polymorphism will completely change the byte sequences of the main program body.

Assembly Code

Machine code and assembly code can be translated to each other through assembler and disassembler. The advantage of assembly code over machine code is that assembly code can be understood by a human expert; therefore, as a kind of feature, assembly code is easier to be preprocessed (e.g., grouped, filtered, truncated etc.) to improve its quality.

Moskovitch et al. [101] state that the malicious engine may locate in different positions in a variants of the same malware so operands of instructions are not stable across executables. Therefore, they propose to drop operands and use only opcodes as features to improve the robustness. From the perspective of feature engineering, it can also relieve

the curse of dimensionality. They extract assembly code by dissembling the executables with IDA Pro. They use term frequency (TF) or TF-IDF of each opcode n-gram (n=1,2,...,6) as the feature representations and use document frequency (DF), information gain ratio, or Fisher score to select informative features. Their best result is achieved using TF values of opcode bigram as features selected by Fisher score. The disadvantage of their method is that it is still prone to dead code insertion, operation transpositions, instruction substitution, packing, and polymorphism.

To counter packing and polymorphism, Dai et al. [42] run malware in a VM and record the sequence of the executed byte code which will be disassembled. To counter dead code insertion, operation transpositions, they use three kinds of two-opcode combinations: unordered opcodes in a block, ordered but not necessarily consecutive opcodes in a block, consecutive opcodes in a block. They use the association between the frequency of a feature in training dataset and a class as the criterion and apply a variant of Apriori [3] to select top L features. Although unordered opcodes in a block and ordered but not necessarily consecutive opcodes in a block improve the resistance to dead code insertion and reorder of operations, those features are too flexible, so they also bring more false positive cases.

Royal et al. [121] aim to unpack hidden code which is encrypted as data at compile-time. They first store the static code of an executable and then execute the file in a VM and check whether each instruction executed is within the stored static code area. If it is not, it is a part of hidden code. They claim that the main malware engine should be in the hidden code if both of them exist and experiment results also illustrate the hidden code enhances the accuracy of ClamAV [6] and McAfee Antivirus [7].

Anderson et al. [7, 8] do not use the presence of each opcode as a feature but use the transition probability from one opcode to another as a feature which is similar to the way they use the byte sequence feature. In their first paper [7], they just extract assembly code by recording the execution of a file to acquire assembly code. In their second paper [8],

[6] http://www.clamav.net/
[7] https://www.mcafee.com/en-us/index.html

they also use IDA Pro to disassemble an executable and the assembly code from the two sources are treated as two independent features. In addition, they group instructions into categories in several granularities according to the functions of the instructions to combat the curse of dimensionality in their second paper [8]. Actually, it also reduces the impact of instruction substitution. In their preliminary experiment, they find if they use instructions with operands, the performance will be worse [8]. Their explanation is that by ignoring operands, they remove sensitivity to register allocation and other compiler artifacts.

Santos et al. [125] disassemble executables and use weighted term frequencies of opcode n-grams as features. The weighted term frequency of an opcode n-gram is the product of the information gain of all opcodes in the n-gram times the normalized TF of the n-gram.

Opcode n-grams are indeed informative features. However, using assembly code in this way causes high loss of information. One reason is that the operands are abandoned; the other is that the order of them is missing since opcode n-grams are just tiny pieces of assembly code. To make an analogy, if we break a sentence into phrases, by reordering the phrases we can create a sentence that has entirely different meaning. Comparably, with a set of opcode n-gram, entirely different programs can be formed. The way to effectively utilize operands and to understand the entire sequence of assembly code has not been proposed yet.

System Calls

System calls provide the services of an operating system to user programs via Application Program Interface (API) which reside in some DLL files. From the services an executable required from an operating system, what behaviors it may intend to do or what it would be able to do can be inferred.

To understand an executable's behavior, Schultz et al. [127] use the following three features:

1. The list of DLLs used by an executable

2. The list of DLL function calls invoked by an executable

3. Number of different function calls invoked by an executable within each DLL

They extract those features from PE headers to understand how resources affect an executable's behavior and how heavily each DLL is used. The first two are used as binary features and the third is a kind of real-valued feature.

Bayer et al. [18] and Santos et al. [125] execute an executable in an emulator to monitor and record the system calls and the corresponding parameters. Then they use those API functions to acquire behaviors of an executable during execution including I/O activity, registry modification activity, process creation/termination activity, network connection activity of an executable, self-protection behavior, system information stealing, errors caused by the execution, and interactions with Windows Service Manager. The behaviors are used as features for further analysis.

Fredrikson et al. [57] also use an emulator to monitor and record system calls. Then they use the relations between system calls and their parameters to form a dependency graph in which nodes are system calls and edges connect system calls sharing some parameters. They define a behavior to be a subgraph of it and behaviors that can be used to distinguish malware from benignware will be mined and used to detect malware.

Anderson et al. [8] and Huang and Stokes [69] group the system calls into high-level categories where each category represents functionally similar groups of system calls, such as painting to the screen, writing to files, or cryptographic functions. Anderson et al. [8] then feed the trace of groups of system calls to a Markov chain so that they use transition probability of system calls to be a kind of feature. Huang and Stokes [69] use the existences of those high-level API call events as binary features.

Islam et al. [72] and Dahl et al. [41] extract Windows API function calls and their parameters by running an executable in a VM. Islam et al. [72] treat Windows API functions and parameters as separate entities and use the occurrence frequency of each entity as

their feature. Dahl et al. [41] use the combination of a single system API call and one input parameter and API tri-grams which consist of three consecutive API function calls as binary features. They also perform feature selection using mutual information.

Learning from Schultz et al. [127], Saxe and Berlin [126] use PE headers to extract the DLL functions an executable invoked. For each invoked DLL function call, they combine the function's name and its DLL file's name and calculate the hash code of the combination to form a histogram. They use the bins' values of the histogram as features. However, this setting is not reasonable since the DLL function calls which have the same hash value are probably functionally and literally irrelevant.

Kolosnjaji et al. [77] use the dynamic malware analysis system Cuckoo sandbox to extract the sequence of the system calls invoked by an executable. They use one-hot representations of them and feed the full sequence of system calls without omitting the order to a sequential deep learning model.

The same as assembly code, system call sequences can also be obfuscated. For instance, malware authors can make an executable invoke some irrelevant system call calls and submerge the system calls they use to fulfill their purpose in them. So this kind of feature is not reliable either.

Control Flow Graphs

A control flow graph is a directed graph to represent the relations of basic blocks. A basic block is a sequence of instructions in the middle of which there is no jump or branch instructions and a directed edge represents jumps in the control flow [8]. Cesare and Xiang [27] state that similarities between malware variants are reflected by variants sharing similar high-level structured control flows. Anderson et al. [8] also find it very difficult for a polymorphic virus to create a semantically similar version of itself while modifying its control flow graph enough to avoid detection. Therefore, they use control flow graphs as features.

Cesare and Xiang [27] find that compressed and encrypted data have relatively high entropy, so they first use entropy of byte sequence to detect whether an executable is packed. If it is, they use an application level emulator to execute the executable. If any newly written memory is executed, then they determine that the packed code begins to be revealed. They use entropy of byte sequence of the executable's memory image to detect the completion of hidden code extraction. After it is completed, the memory image of the binary is disassembled using speculative disassembly [81]. Finally, they use the process of structuring to recover high-level structured control flows from control flow graphs of procedures and represent them using strings of character tokens. The strings representing control flow graphs are all saved as signatures.

Anderson et al. [8] extract control flow graphs from static assembly code acquired by IDA Pro as features. Specifically, they use the occurrence frequencies of k-graphlets (a subgraph of k nodes) of control flow graphs as features.

To counter the detection using control flow graphs, malware authors can use control flow flattening and bogus control flow obfuscation techniques to change the control flow without affecting the functionality so this would reduce the effectiveness of control flow graph features.

Function

Islam et al. [72] and Chen et al. [30] use function level features for malware analysis.

Islam et al. [72] find function lengths contain statistically significant information to distinguish malware families. So after they get the assembly code of each executable with IDA Pro, they calculate the length of them measured by the number of bytes and use the occurrence frequencies of function lengths as features. However, function length is obviously the least robust feature against obfuscation. Function length can be arbitrarily increased by inserting dead code or decreased by splitting them into multiple functions.

Two functions which are semantically similar to each other are considered to be a 'clone' of each other. Chen et al. [30] assume files belonging to the same malware family

share some functions which are connected using clone relation. So, they cluster functions to groups in which any two functions can be connected directly or indirectly using the clone relation and pick one function from each group as an exemplar to be a signature. They use NiCad [39] to detect whether two functions are clone to each other. Although their system works on Android APK files, the methodology can be directly applied to classifying executable malware. However, to use one exemplar function to represent a group of functions is problematic. As the same function evolves over generations, the newest version may appear quite different from the original version. If the oldest version is picked as the exemplar, the clone detector tend to fail to identify some unknown new generation of it.

Miscellaneous File Information

Some of the miscellaneous file properties can help to distinguish malware from benignware since the average or majority values of them are significantly different between the two groups [8]. Anderson et al. [8] use file size, entropy, packed or not, number of static/dynamic instructions, and number of vertices and edges in control flow graph as features. Saxe and Berlin [126] use the values of all numerical fields of PE files extracted with "pefile" Python parsing library[8] as features. Those features may be helpful but apparently not informative enough.

Summary

According to the presented analysis, either the effectiveness of the mentioned usages of all the features can be diminished somehow or they are not informative enough. Therefore, in some papers multiple features are used simultaneously [8, 41, 69, 72, 125–127]. The intuition is that any single feature source can be obfuscated to evade the detection but it is extremely difficult to obfuscate all features simultaneously without hindering the functionality [8, 72].

[8] https://github.com/erocarrera/pefile

2.2.4 Classification Models

In this section, we introduce the classification models which take the features as input and predict an executable's class. The models are categorized as traditional machine learning models, deep learning models, association mining, graph mining and concept analysis, approximate dictionary search, and function clone detection.

Traditional Machine Learning Models

The most popular traditional machine learning models for malware analysis include *Naive Bayes classifier (NBC)*, *decision tree (DT)*, *K-nearest neighbors (KNN)*, and *support vector machine (SVM)*. Some papers use multiple traditional machine learning models and their main contributions are on the feature extraction technique, choice of features, and feature representations [42, 72, 78, 101, 125, 127]. We first briefly introduce the machine learning models below.

Naive Bayes Classifier. A Naive Bayes Classifier [123] uses Bayes' theorem to calculate the conditional probability of a sample belonging to a class given the input features. It can be formally described in the following equation:

$$P(C_i|x) = \frac{P(x|C_i)}{P(x)}P(C_i) \tag{2.1}$$

where x is a sample with its features and C_i is a class. It is based on the naive Bayes conditional independence assumption that all the features are independent to each other given the class it belongs to:

$$P((x_1, x_2, ..., x_n)|C_i) = P(x_1|C_i)P(x_2|C_i)...P(x_n|C_i) \tag{2.2}$$

where x_i is a feature of x. Although the assumption do not hold in many occasions, the prediction results are good. Its advantage is that the prediction result is explainable because the degree of contribution of each attribute can be measured and visualized.

Decision Tree. A DT classifier [114] uses a tree structure to represent the classification process. Internal nodes of a DT are tests on the values of features and edges correspond to a choice on the values of a variable. Leaf nodes represent the final class of samples fall into it. The tree structure is constructed based on the informativeness of each feature such as information gain ratio and Gini index conditioned on the current choices. A DT is also an interpretable classifier since a DT can be translated to a set of if-else-then rules.

K-Nearest Neighbor. KNN [6] is an instance-based classifier. It predicts the class label of an unknown sample to be the (weighted) majority vote of the classes of its K nearest neighbors in the training set.

Support Vector Machine. An SVM [19] is a binary classifier that determine a hyperplane to separate samples from two classes with the largest margin. An essential characteristic of an SVM is that it can utilize kernel trick to map samples from the original feature space to a high-dimensional (even infinite) feature space to perform non-linear classification.

Bayesian Network. A *Bayesian network (BN)* [74] is a probabilistic graphical model which is also based on Bayes' theorem. In the graphs, variables are represented as vertices and the dependencies as directed edges. The graph is used for the inference of value of any variable (the class of a sample in our case) in it based on the information of other variables (features).

RIPPER. RIPPER [38] is a rule-based classifier which builds a set of rules to classify samples while minimizing the error of the number of misclassified training samples.

Schultz et al. [127] apply three models on their three feature sets respectively. They apply RIPPER on DLL an executable used, DLL function calls, and number of different function calls invoked within each DLL separately; apply NBC on string features; separate byte sequence features into multiple groups and apply one NBC on each group and the prediction is votes of all those NBCs. The reason why they separate byte sequence

features is that the set of byte sequences is too large and the memory cannot hold all byte sequence features to train one model. Both NBC and multiple NBC achieve 97% accuracy on a dataset of 4,266 samples which is better than RIPPER.

Kolter and Maloof [78] apply KNN where $K = 5$, NBC, DT, boosted NBC, and boosted DT on their top 500 byte sequence n-grams. The boosting algorithm they use is Adaboost.M1 [58]. The best area under the Receiver Operating Characteristic curve (auROC) is achieved by boosted DT: 0.9958 on a dataset of 1,037 samples, 0.9958 on a dataset of 3,622 samples.

Moskovitch et al. [101] apply DT, NB, boosted DT, boosted BNB, and artificial neural networks (ANN) on TF or TF-IDF of opcode n-grams. The boosting algorithm they use is also Adaboost.M1 [58]. They fail to give the specific structure of their ANN. The boosted DT achieves the best accuracy: 94.43% on a dataset of 26,093 samples. They also find among the models they have tried, only ANN is prone to the imbalance of malware percentage of the dataset.

Dai et al. [42] apply DT and SVM on their feature sets respectively: unordered opcodes in a block, ordered but not necessarily consecutive opcodes in a block, consecutive opcodes in a block. The best result is achieved by SVM with ordered but not necessarily consecutive opcodes in a block which is 91.9% on a dataset of 635 samples.

Nataraj et al. [106] apply KNN where $K = 3$ with Euclidean distance to classify executables into their corresponding families based on texture features of their created malware image. They achieve 99.93% accuracy on a dataset of 1,713 malware files from 8 families and 97.18% on a dataset of 9,458 malware files from 25 families.

Anderson et al. [7] apply the kernel trick to an SVM model to classify executables. Their feature is a matrix in which each entry is the transition probability from one opcode to another. They use a linear combination of two kernels for the feature. One is a Gaussian kernel on the entries of the matrix which is aimed to search for local similarities between

two transition probability matrices x and x':

$$K_G(x, x') = \sigma^2 e^{-\frac{1}{2\lambda^2}\sum_{i,j}(x_{ij}-x'_{ij})^2} \tag{2.3}$$

where x_{ij} and x'_{ij} are transition probabilities from opcode i to opcode j in the code of two executables. The other kernel is a Gaussian kernel on top-k eigenvectors of the Laplacian matrix [36] created from the transition probability matrices:

$$K_S(x, x') = \sigma^2 e^{-\frac{1}{2\lambda^2}\sum_{k}(\phi_k(L(x))-\phi_k(L(x')))^2} \tag{2.4}$$

where $\phi_k(L(x))$ and $\phi_k(L(x'))$ are the k-th eigenvectors of Laplacian matrices. Since the eigenvectors encode global properties, this kernel searches for similarity of global structure. With the combined kernel:

$$K(x, x') = \mu K_G(x, x') + (1 - \mu)K_S(x, x') \tag{2.5}$$

where $0 \leq \mu \leq 1$, the SVM achieves the best accuracy: 96.41% on a dataset of 2,230 samples.

Comparing with their earlier work [7], Anderson et al. use more features in their later work [8]. They apply Gaussian kernel to features based on Markov chain (i.e., transition probabilities) and miscellaneous file information. The features based on Markov chain include byte sequences, static opcode sequences, dynamic opcode sequences, and system call traces. They apply a graphlet kernel to control flow graph feature. Let f_G be a vector representing the occurrence frequency of each k-graphlet in G. The graphlet kernel is as follows:

$$D_G = \frac{f_G}{number\ of\ all\ graphlets\ of\ size\ k\ in\ G} \tag{2.6}$$

$$K_g(G, G') = D_G^T D_{G'} \tag{2.7}$$

They combine all the kernels with a weight to each of them and iteratively learn the weight and parameters of each kernel until convergence. They achieve the best accuracy with all features and kernels which is 98.07% on a dataset of 1,556 samples.

Islam et al. [72] apply SVM, DT, KNN, and random forest (RF) which is an ensemble method based on DT on all three features: function length, printable strings, and names and parameters of windows API function calls. The RF achieves the best accuracy: 97.055% on a dataset of 2,939 samples.

Santos et al. [125] apply DT, RF, KNN, BN, SVM on their statically extracted weighted term frequencies of opcode n-grams and dynamically extracted behavior features. SVM with normalized polynomial kernel achieves the best accuracy: 96.60% on a dataset of 2,000 samples. With only statically extracted opcode n-gram features, the model achieves comparable accuracy: 95.90%. With only dynamically extracted behavior features, the model achieves 77.26% accuracy. It means that their static features are more effective than the dynamic features.

Deep Learning Models

Dahl et al. [41] apply a deep learning model on their 179,000 binary features. The first layer is a random projection layer which maps the input features to a much lower dimensional space (4000 dimensions). The difference between the random projection layer and a standard fully connected layer is that the weight of the projection matrix is not updated during training. The entries of it are sampled by following an independent and identically distribution over {-1,0,1}. On top of that, they apply one to three fully connected layers with sigmoid activation functions and a 136-way softmax layer as output. The 136 classes include 134 important malware families, a generic malware class, and the benignware class. They also try using a Gaussian-Bernoulli restricted Boltzmann machine (RBM) to pre-train the hidden layers. The best result is achieved by the model with 1 hidden layer without pre-training which is 9.53% test error rate for multiclass classification and 0.49%

test error rate for binary classification on a dataset of 2.6 million samples. They also find the random projection performs better than Principal Component Analysis (PCA).

Saxe and Berlin [126] propose a deep feed-forward neural network consisting of three fully connected layers, where the dimensions of the two hidden layers are 1024 followed by a dense layer to get the output. They apply dropout to the input and the two hidden layers to prevent overfitting. The activation functions of the two hidden layers are parametric rectified linear units (PReLU) to yield an improved convergence rate without loss of performance and the activation function of the output layer is sigmoid. They also use Bayesian Calibration to calculate the unbiased probability that an executable is malware. They achieve a detection rate of 95.2% and a false positive rate of 0.1% on a dataset of 431,926 samples.

Huang and Stokes [69] propose a neural network for multi-task training. One task is malware detection to predict whether an unknown executable is malicious or benign and the other is a 100-class classification task. The 100 classes include 98 important malware families, a generic malware class, and the benignware class. Learning from Dahl et al. [41], Huang and Stokes [69] also use a random projection layer to reduce the dimension to 4,000 from 50,000 and they normalize each of the 4,000 dimension to be zero mean and unit variance. Then they use multiple hidden fully connected layers with dropout and RELU activation. On top of them there are two single layers for each of the two classification tasks. The final loss function is a weighted summation of each of the individual loss functions. Experiment results show that multi-task learning only improves the performance of malware detection and harm the performance of malware classification in most experiment settings. Specifically, the setting to achieve the best result for malware detection: 0.3577% test error is using two hidden layers in the multi-task learning framework. The setting to achieve the best result for malware classification: 2.935% test error is to use one hidden layer in either single task or multi-task learning situation.

Kolosnjaji et al. [77] propose a combination of a convolutional neural network (CNN) and a Long Short-Term Memory (LSTM) network [68] to predict the family of an exe-

Table 2.1: Summary of features and models used in malware analysis systems.

Paper	Year	Features	Models
Schultz et al. [127]	2001	(s) DLLs/DLL functions (s) Printable strings (s) Byte sequence	RIPPER NBC multiple NBC
Kolter and Maloof [78]	2004	(s) byte sequence n-gram	KNN, NBC, boosted NBC, DT, boosted DT
Bayer et al. [18]	2006	(d) system calls→6 groups of behaviors	N/A
Royal et al. [121]	2006	(d) assembly code	ClamAV and McAfee
Moskovitch et al. [101]	2008	(s) opcode n-gram	DT, NB, boosted DT, boosted BNB, ANN
Dai et al. [42]	2009	(d) unordered opcodes in a block, ordered but not necessarily consecutive opcodes in a block, consecutive opcodes in a block	DT, SVM
Fredrikson et al. [57]	2010	(d) system calls and parameters	Graph Mining and Concept Analysis
Ye et al. [147]	2010	Windows API calls	HAC
Cesare and Xiang [27]	2010	(d) control flow graph	BK Trees
Nataraj et al. [106]	2011	(s) texture of byte sequence image	KNN
Anderson et al. [7]	2011	(d) transition probability of opcode	SVM
Anderson et al. [8]	2012	(s) transition probability of bytecode (s)(d) transition probability of opcode (s) k-graphlet control flow graph (d) system call (s) misc.	SVM
Dahl et al. [41]	2013	(d) string (d) system call tri-grams (d) system call+parameters	random projection + feed-forward neural network
Islam et al. [72]	2013	(s) Function length (s) string (d) API function/parameters	SVM, DT, KNN,RF
Santos et al. [125]	2013	(s) opcode n-gram (d) system calls→7 groups of behaviors	DT, RF, KNN, BN, SVM
Chen et al. [30]	2015	(s) function	function clone detection
Saxe and Berlin [126]	2015	(s) string (s) byte sequence (s) DLL functions (s) misc.	feed-forward neural network with Bayesian Calibration
Kolosnjaji et al. [77]	2016	(d) system call sequence (ordered)	CNN+LSTM
Huang and Stokes [69]	2016	(d) string (d) Windows API calls	random projection + multiple hidden layers +2 separate output layers

(s) represents that the corresponding kind of feature is statically extracted.

(d) represents the corresponding kind of feature is dynamically extracted.

cutable using the dynamically extracted system call sequence. They first use two convolution layers to capture the correlation between consecutive API calls and then apply max-pooling to reduce the dimensionality. The output sequence is fed to an LSTM layer to model the sequential dependencies of API calls. Then a mean-pooling layer is used to extract features of the highest importance from the LSTM's output and reduce the complexity for further data processing. They also use dropout to prevent overfitting and a softmax layer to output the probability of each class. Their proposed deep learning model significantly outperforms feed-forward neural networks, CNN, SVM, and Hidden Markov Model and achieves 85.6% on precision and 89.4% on recall on a dataset of 4753 samples. The advantage of their model is that it can fully utilize the order of system calls which may also be a drawback if the system call sequence is obfuscated. One problem of their model is they use mean-pooling rather than max-pooling to extract features of highest importance produced by the LSTM is not quite reasonable.

Associative Classifier

An associative classifier relies on association rules that can be used to distinguish samples between two classes to perform classification. It is a special case of association rule mining where only the class of a sample can be the consequent (a.k.a. right-hand-side) of a rule. Ye et al. [147] propose to use hierarchical associative classifiers (HAC) to classify executables in a gray list based on Windows API calls. Normally, a gray list has an imbalanced class distribution issue: most samples are benignware. To handle this issue, they use one associative classifier to achieve high recall and the other to achieve high precision. There are three techniques regarding creation of an associative classifier: 1) use FP-Growth algorithm to find candidate association rules (i.e., discriminative combinations of Windows API calls) 2) prune the candidate rules based on χ^2, data coverage, pessimistic error estimation, and significance w.r.t to its ancestors 3) reorder rules: first rank the rules whose confidence values are 100 by confidence support size of antecedent (CSA) and then reorder the remaining rules by χ^2 measure. Using those three techniques, they create a

two-level associative classifier to detect malware from a gray list labeled by a signature-based anti-virus engine. The first-level associative classifier is aimed for a higher recall of malware. It only keeps the rules related to benignware with 100% confidence and the rules related to malware with confidence greater than a pre-defined threshold; then it uses the rule pruning technique to reduce the generated rules and build the classifier; finally "Best First Rule" method is used to predict samples from the gray list. The samples labeled to be malware by the first associative classifier are fed to the second level associative classifier which is aimed at optimizing the precision. It works with the following steps: 1) select those samples whose prediction rules of malware have 100% confidence values, marking them as "confident" malware; 2) ranking the remaining files labeled to be malware in a descending order based on their prediction rules' χ^2 values; 3) select the first k files from the remaining ranking list and marking them as "candidate" malware; 4) mark the remaining (unselected) files as "deep gray" files. Experiment results show the proposed HAC is effective even for an extremely imbalanced dataset. In addition, HAC is also an interpretable classifier which can be easily translated to simple if-then rules.

Graph Mining and Concept Analysis

Fredrikson et al. [57] extract behaviors (dependency graphs of system calls and their parameters) that can distinguish malware from benignware using structural leap mining [144]. Then they use the behaviors to form discriminative specifications. A specification is a collection of behaviors and a characteristic function that defines one or more subsets of the collection. A program matches a specification if it matches all of the behaviors in at least one characteristic subset. A specification is entirely discriminative if it matches malicious programs but does not match benign programs. They use formal concept analysis [138] to create candidate discriminative subsets of behaviors and use Simulated Annealing algorithm [20] to choose from all the created candidate subsets of behaviors to form an approximate optimal specification which has true positive larger than a threshold and lowest false positive among all specifications which have larger true

positive rate than the threshold. During the test period, if a program matches a specification, it will be classified to be malware. The created specification achieves 86% true positive rate and 0 false positive rate on a dataset of 961 samples.

Approximate Dictionary Search

Cesare and Xiang [27] detect variants of malware in a database using the ratio of procedures an unknown executable share with malware in the database. They first convert the control flow graphs of each procedure in an unknown executable to character strings in the same way they create signatures and each procedure is assigned a weight using the length of the string: $weight_x = \frac{len(s_x)}{\sum_i len(s_i)}$, where s_x is the length of the string x. Then they use BK Trees [12] to retrieve the strings in the signature database which have less Levenshtein distance with strings representing procedures of the target file than a threshold. The similarity ratio of two strings is calculated as $w_{ed} = 1 - \frac{ed(x,y)}{max(len(x),len(y))}$. Similarity ratios of procedures from the unknown executable and a malware sample in the database are accumulated proportional to the weights of one of the procedures. As there are two weights (one is the weight of a procedure of the unknown executable, the other is the weight of a procedure of a malware sample in the database), the similarity between the unknown executable and a malware sample in the database is the product of two asymmetric similarities: a similarity that identifies how much of the unknown executable is approximately found in the malware in the database, and a similarity to show how much of the malware in the database is approximately found in the unknown executable. If the similarity of the unknown executable to any malware in the database equals or exceeds a threshold of 0.6, then it is deemed to be a variant. Experiment results show that this method can successfully identify variants of malware. However, since they use a symmetric similarity calculated as the product of two asymmetric similarities, it can not handle asymmetric situations. For instance, if a very large unknown executable contains the whole program of a malware sample in the database but that malicious program only

take up 1% of the whole content of the unknown executable, the similarity calculated in their way would still be small and the model will fail to recognize that it is malware.

Function Clone Detection

Chen et al. [30] uses NiCad [39] to detect whether an APK file contains any function that is a clone of an exemplar function which represents a signature of a malware family. If a match is found, the file is predicted to be an instance of that malware family. They achieve 96.88% accuracy on a dataset of 1170 APK files from 19 malware families.

Summary

Although almost all malware classification models mentioned in this section achieve more than 85% on accuracy, auROC, true positive rate, etc., the experiment results of different malware classification models are not comparable with each other since the datasets they use are different. In several papers, multiple traditional machine learning models are used [42, 72, 78, 101, 125, 127], but there is no model outperforms others in all of those papers. What we observe is that either (boosted) DT or SVM achieves the best results in those papers. We summarize the features and classification models of malware analysis systems in Table 2.1.

2.3 Adversarial Attack and Defense

In this section, we present a literature review on adversarial attack and defense techniques.

In 2013, Szegedy et al. [134] showed that a hardly perceptible perturbation on an image, which is found by maximizing a neural-network-classifier's prediction error, can lead to its mis-classification. This property could be utilized by an adversary to cause severe security issues. For example, an autonomous vehicle that fails to recognize a stop sign may cause a traffic accident; failure to recognize a malware program may cause a

huge loss to a computer user. Consequently, adversarial attack and defense techniques are being extensively studied [31, 87, 142].

2.3.1 Definitions

We formally define adversarial attack and defense in this subsection.

Adversarial Attack

A machine learning model is a function $f(\boldsymbol{x})$ that maps an input sample $\boldsymbol{x}$ to an output $\boldsymbol{y}$, which is a vector of n dimensions, where n is the number of classes. Each dimension of $\boldsymbol{y}$ corresponds to the probability that $\boldsymbol{x}$ corresponds to a certain class. The objective of adversarial attacks is to add a minimal perturbation ϵ to a natural sample $\boldsymbol{x}$ so that $f(\boldsymbol{x} + \boldsymbol{\epsilon})$ presents a false result, i.e., the class with the largest probability is not the real one. $x' = \boldsymbol{x} + \boldsymbol{\epsilon}$ is called an *adversarial sample*.

Adversarial Defense

An adversarial defense mechanism is to achieve two objectives by modifying $f(\boldsymbol{x})$ to be $f'(\boldsymbol{x})$. The first objective is to minimize the changes made by a perturbation ϵ: $min\{|f'(\boldsymbol{x}+\boldsymbol{\epsilon}) - f'(\boldsymbol{x})\}|$. The second is to minimize the changes to the prediction of natural (unperturbed) samples: $min\{|f'(\boldsymbol{x}) - f(\boldsymbol{x})|\}$.

2.3.2 Taxonomy

In this subsection, we categorize adversarial attack and defense from different perspectives.

Attack Phase

Adversarial attacks can be classified as *poisoning attacks* and *evasion attacks*, based on the phase in which an attack is performed. A poisoning attack is performed in the training

phase of the objective classifier [140]. The attacker injects some fake samples into the training set of a classifier, causing the classifier to consistently mis-classify samples with certain characteristics that are similar to the injected fake samples. An evasion attack is performed after a classifier is trained; a legitimate sample is perturbed to cause the classifier to mis-classify it. This kind of attack is more practical for attackers to perform and has been extensively studied. Therefore, it is also our focus.

Adversarial Objectives

Based on the objectives, attacks can be categorized as *targeted attacks* or *untargeted attacks*. A targeted attack is aimed at leading a classification system to classify an adversarial sample to an objective class. An untargeted attack is aimed at leading a classification system to classify an adversarial sample to any incorrect class. In both cases, the adversarial sample should be crafted from a correctly classified natural sample.

Attack and Defense Settings

Adversarial attack and defense can be conducted in white-box, black-box, and gray-box scenarios, depending on how much an adversary knows the target model to attack.

White-box Setting. In the early stage of research in adversarial attack and defense, an adversarial attack setting is one in which an adversary has full knowledge of the target model $f(\boldsymbol{x})$, including its architecture and parameters, to craft adversarial samples. This is called the *white-box setting*.

Black-box Setting. In a *black-box setting*, an adversary has no knowledge of the target model, but can request its output $y = f(\boldsymbol{x})$ for any valid input $\boldsymbol{x}$. In a black-box setting, there are two special situations, namely the *label only setting* and the *partial-information setting*. In a label only setting, the output y is the predicted class without probability. In

a partial-information setting, the attacker only has access to the probabilities of a sample belonging to the top-k classes [70].

Gray-box Setting. In addition to the white-box and black-box settings that most studies focus on, there is also a gray-box (a.k.a., semi-white, semi-black) setting. An adversary has partial knowledge of the system to be attacked. There are multiple situations in which researchers are interested:

1. The adversary knows the architecture of the model to be attacked, but does not have access to its parameters and training set.

2. The adversary knows the details of the model to be attacked, without knowing the existence of a defense mechanism.

2.3.3 Adversarial Attack Methods

Below, we introduce adversarial attack methods in both white-box and black-box settings.

White-box Setting

We categorize white-box attacks as *explicit perturbation* and *implicit perturbation* methods. Explicit perturbation methods directly compute the perturbation to be added to a natural sample. Implicit perturbation methods use a neural network to compute an adversarial sample, which is equivalent to implicitly adding a perturbation onto the natural sample. Below are the typical explicit perturbation methods.

In 2014, Goodfellow et al. [61] proposed the *fast gradient sign method (FGSM)* to craft adversarial samples. They perturb each sample only once toward the gradient sign direction according to the following equation:

$$\eta = \epsilon sign(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y)) \tag{2.8}$$

where $J(\boldsymbol{\theta}, \boldsymbol{x}, y)$ is the objective function to train the targeted neural network to correctly classify $\boldsymbol{x}$. It is the most efficient direction to change the prediction. This is the most primitive and straightforward method to craft adversarial samples; however, this one-step-perturbation may not be enough to craft an adversarial sample.

As an extension to FGSM, in 2016, Kurakin et al. [82] proposed an *iterative gradient sign (IGS)* to craft adversarial samples through multiple perturbation steps. In each step, IGS perturbs the pixels toward the gradient sign direction and clips the perturbed sample to stay within the l_∞-ball neighbourhood of the original sample:

$$\boldsymbol{x}_{n+1}^{adv} = Clip\{x_n^{adv} + \alpha sign(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))\} \tag{2.9}$$

Similarly, Madry et al. [95] also extended FGSM to a multiple step procedure to form a method known as *projected gradient descent (PGD)*. The difference between PGD and IGS is that IGS begins by perturbing a natural sample and PGD, by perturbing a random point in the l_∞-ball around the natural sample.

The aforementioned two methods perturb all dimensions of a sample simultaneously. Differently, Papernot et al. [109] proposed the *forward derivative method (FDM)*. It evaluates the sensitivity of the output to each input dimension using the Jacobian matrix of the model. Then, it constructs an adversarial saliency map based on the Jacobian matrix indicating which input feature should be perturbed in each iteration of perturbation, until it reaches an adversarial sample.

Moosavi-Dezfooli et al. [100] proposed *Deepfool*, which attacks a system by pushing a sample toward the orthogonal vector of the target classification boundary:

$$\boldsymbol{r}_*(\boldsymbol{x_0}) := argmin||r||_2 \tag{2.10}$$

$$subject\ to\ sign\ (f(\boldsymbol{x_0} + \boldsymbol{r})) \neq sign(f(\boldsymbol{x_0})) \tag{2.11}$$

Such a perturbation is the optimally efficient perturbation in terms of L_2 norm.

Carlini and Wagner [23] also took the adversarial attack as an optimization problem. The objective function includes the distance between the adversarial sample and its original sample, as well as the difference between the probabilities of the target class and the real class:

$$minimize\ c\dot{|}|x - x'||_2^2 + loss_{F,l}(x') \tag{2.12}$$

Thus, in its objective function, both the purpose to mislead a classifier and to minimize the perturbation are considered.

Athalye et al. [11] proposed a white-box attack method called *Backward Pass Differentiable Approximation*. This method breaks adversarial defenses based on gradient masking. They summarize that the gradient obfuscation is achieved by using non-differentiable operations, employing randomized transformations to the input, and/or using optimization loops. To cancel out the gradient obfuscation, during the backpropagation phase, they approximate non-differentiable operations with differentiable operations, apply Expectation over Transformation, and reverse the optimization loop with reparameterization. They find that they can thus totally evade six out of seven gradient masking-based defense methods by combining these techniques. Below are implicit perturbation methods.

Baluja and Fischer [15] proposed to use a neural network called *Adversarial Transformation Networks (ATNs)* that takes a natural sample as its input and transforms it into an adversarial sample. It is trained with an objective function of two terms:

$$L = \sum_{x_i} [\beta L_x(g(x_i), x_i) + L_y(f(g(x_i)), f(x_i))] \tag{2.13}$$

where $g(x)$ is the ATN, $f(x)$ is the target network to mislead, $\sum_{x_i} \beta L_x(g(x_i), x_i)$ is to minimize the scale of perturbation, $L_y(f(g(x_i)), f(x_i))$ is to mislead the target neural network, and β is a weight to balance those two objectives.

Similarly, Yu et al. [148] proposed to use a *conditional generative adversarial network (CGAN)* to craft adversarial samples. They train the CGAN network with two objectives:

$$L = L_{CGAN} + \alpha L_{fool} \tag{2.14}$$

where L_{CGAN} is the standard CGAN reconstruction objective to generate a sample resembling the original sample, and αL_{fool} is the objective to fool the targeted neural network. α is a weight to balance those two objectives.

Bai et al. [13] proposed a GAN-based network called *Attack-Inspired GAN (AI-GAN)* to generate adversarial samples. Their discriminator has two objectives, namely to discriminate between real and perturbed images, and to classify an image to its correct class. The generator is aimed at generating a fake image to trick the target model and the discriminator.

Black-box Setting

In a black-box setting, attacks are conducted based on the information acquired by querying the target classification system. The following are the two main types of black-box attack methods.

The first type can be addressed as *Substitute Model* attacks. The concept of substitute model attack was proposed by Papernot et al. [108]. They find that adversarial samples have some transferability, i.e., a sample that compromises one model is likely to compromise another. Hence, they propose to obtain the outputs corresponding to a set of synthetic samples from the target model and use them to train a substitute model. They craft adversarial samples by attacking the substitute model in the white-box setting and expect that those samples can mislead the target model. Recent works [32, 70, 95, 104, 105] have shown that the transferability of adversarial examples crafted from a substitute model is limited and can be defended effectively.

The other type of attack can be addressed as *Perturbation Trial* attacks. Since the output of the target model can be acquired by the adversaries, they can try perturbing certain input dimensions by increasing and decreasing the values and check whether either of them would lead the prediction toward their objective direction, e.g., the probability of the target class increases, or the probability of the real class decreases. Essentially, instead of using an analytical solution to acquire the gradient in the white-box setting, this solution in a black-box setting uses numerical differentiation to achieve the same purpose. This technique is also widely used in gradient checking for validating the implementation of back propagation for training neural networks [139]. The adversary repeats that procedure multiple times and accumulates effective perturbations, until an adversarial sample is found or a maximum number of trials has been performed.

In 2017, Chen et al. [32] proposed the *Zeroth Order Optimization (ZOO)* black-box attack method, which is a coordinate-wise gradient estimation:

$$g_i := \frac{\partial f(x)}{\partial x_i} \approx \frac{f(x + he_i) - f(x - he_i)}{2h} \tag{2.15}$$

It perturbs one variable each time and accumulates the perturbations until a successful adversarial sample is found or a preset number of iterations has been tried.

In 2019, Chen et al. [135] extended ZOO to *Autoencoder-based Zeroth Order Optimization Method (AutoZOOM)*. It uses an autoencoder to map a natural sample to a compact representation and perturbs on that representation. Instead of a coordinate-wise gradient estimation, they perform a random full gradient estimation:

$$\boldsymbol{g} := b \cdot \frac{f(\boldsymbol{x} + \beta \boldsymbol{u}) - f(\boldsymbol{x})}{\beta} \cdot u \tag{2.16}$$

They use an averaged random gradient estimation as follows to control the error in the gradient estimation:

$$\hat{g} = \frac{1}{q}\sum_{j=1} g_j \tag{2.17}$$

In 2018, Ilyas et al. [70] proposed to use *Natural Evolution Strategies (NES)* to estimate the gradient for black-box attacks. It is a derivative-free optimization method. To handle a partial information setting, they begin with perturbing a natural sample that belongs to the target class, rather than perturbing the target sample. To handle a label-only setting, they propose to use the scores for perturbed samples that are computed based on the ranking of different class labels, to serve as a proxy for class probability.

2.3.4 Adversarial Defense Methods

Adversarial defense methods can be generally categorized as *proactive defense* or *reactive defense* methods [97].

Proactive Defense

With proactive defense methods, the machine learning models per se are fortified to be robust against adversarial attacks, and there is no independent defense component outside of the machine learning model.

Szegedy et al. [134] and Goodfellow et al. [61] suggested an effective defense method known as *adversarial training*. They find that training a neural network with a dataset including adversarial samples will make the model resistant to adversarial examples. Some later studies find that adversarial training could be effective against certain types of adversarial attacks, but is limited against other types [111].

Guo et al. [64] proposed to transform input images to counter adversarial attacks in a model-agnostic manner. The transformations include image cropping and rescaling,

bit-depth reduction, JPEG compression, total variance minimization, and image quilting. They find that after those transformations, the effects of perturbations are canceled out.

Cisse et al. [37] proposed *Parseval networks* to defend against adversarial attacks. It is a layer-wise regularization method for reducing a network's sensitivity to small perturbations. They achieve the purpose by controlling the global Lipschitz constant of each hidden layer. They find that with a small global Lipschitz constant (e.g., less than 1), the robustness against adversarial perturbations is improved.

Xie et al. [141] claimed that adversarial perturbations impose noise on the features that are learned by neural networks, so that they mis-classify those samples. Therefore, they propose a *denoising block* that is applied at intermediate layers of convolutional networks. The denoising block includes a denoising operation, a 1×1 convolutional layer, and a residual connection layer. The denoising operation could be four different forms, namely non-local means, bilateral filter, mean filter, and median filter. They combine their denoising block with adversarial training to work together against adversarial attacks.

Dhillon et al. [48] proposed a defense mechanism called *Stochastic Activation Pruning (SAP)*. It incorporates randomness to the network by stochastically pruning a subset of the activations in each layer, preferentially retaining activations with larger magnitudes. Meanwhile, the surviving activations are normalized. It is a post hoc defense mechanism that can be applied after a neural network is trained.

Papernot et al. [110] proposed a defensive mechanism called *defensive distillation*. Distillation [67] was originally proposed for the purpose of transferring the knowledge of one or more complex neural networks to a simpler neural network and making the latter achieve similar or better classification performance. Papernot et al. [110] found that since the knowledge extracted during distillation reduces the amplitude of network gradients exploited by adversaries to craft adversarial samples, the neural networks trained with defensive distillation are less sensitive to adversarial samples.

Reactive Defense

Reactive methods defend a classification system by recognizing adversarial samples. Feinman et al. [54] observed that adversarial samples lie off the manifold of natural samples. Based on this observation, they propose to differentiate adversarial samples from natural samples with two methods. One is to use their difference on a kernel density (KD) estimation on the feature space of the last layer of the target neural network. The other is to use Bayesian uncertainty estimates that can identify adversarial samples since they lie in low-confidence regions of the input space to neural networks trained with dropout.

Grosse et al. [62] added a new class for adversarial samples to existing classes and trained a machine learning model to classify the $K + 1$ classes where K is the number of natural classes. Thus, the adversarial samples are added to the training set as the additional class, and the model is trained to identify them.

Lu et al. [93] proposed to quantize the output of RELU activations at some set of thresholds to generate a discrete code and apply the RBF-SVM model on the discrete code as its features to identify adversarial attacks.

Following a different logic, Xu et al. [143] applied deep neural networks on two kinds of features: squeezed and unsqueezed features. They find that the predictions with these two kinds of features are quite different on adversarial samples, which is not the case for natural samples. Therefore, they propose to check the difference between predictions with squeezed and unsqueezed features of a sample to determine if it is an adversarial sample. Those methods could defend against substitute model attacks in their experiments, but their performance on defending perturbation trial attacks is limited [11].

Chapter 3

Interpretable Classification

In recent years, high classification performance has been achieved by increasing classification models' complexities in the applied machine learning community. State-of-the-art deep learning models have many layers, and it is hard to explain their predictions [21,73,118]. Reversely, simple models such as logistic/softmax regression and decision trees have excellent interpretability. There is a commonly known trade-off between classification performance and interpretability [91]. That being said, there have been efforts to propose interpretable complex models and post-hoc methods to explain complex models [43,122]. They come in different forms and at different costs, partly because there is no fixed mathematical definition of the term interpretability, which can be categorized as either local or global. If an explanation is given for the result of an individual sample, it is local; otherwise, it is global and aimed at interpreting the entire model behavior.

Our ultimate objective is to gain trust in the classification results and to gain insights into the target samples, as opposed to gaining understanding of the classification model itself. Hence, we aim at interpretability for local explanation. The overall objective of this study is to propose a neural network architecture that has the following qualities for classification:

- **High classification performance**. The classification performance in terms of accuracy is still the primary quality we expect from the model.

- **Interpretability**. The model is expected to provide accurate explanation for each classification result.

- **Efficiency of interpretability**. This efficiency measures the extra computation that is required to explain a classification result. Previous explanation methods may need to run the model multiple times to explain one prediction. The most ideally efficient scenario is an intrinsically interpretable model that only needs to run once to obtain a classification result, with the explanation as a byproduct.

To achieve these objectives, we propose an intrinsically *interpretable feedforward neural network (IFFNN)* architecture that can achieve classification performance similar to their non-interpretable (i.e., normal) feedforward neural network counterparts and provide accurate explanations with little extra overhead. The contributions of this work are summarized as follows:

- We propose an intrinsically interpretable feedforward neural network architecture that is compatible with any type of feedforward neural network that takes any tensors of a fixed shape as its input for both binary and multi-class classification.

- We conduct comprehensive experiments to evaluate the classification performance and interpretability of the IFFNNs. We compare the classification accuracy of IFFNNs with their non-interpretable counterparts to show that they have similar classification performance, and the IFFNNs have the advantage in terms of interpretability.

- We propose a synthetic interpretability benchmark dataset to evaluate the interpretability of classification models. It can generate an unlimited number of samples with the reasons why they belong to a specific class.

3.1 Interpretable Feedforward Neural Network

The explanation expected in this study is defined as follows:

Definition 3.1.1 (Explanation). Let a sample be a p-th-order tensor $\boldsymbol{X} \in \mathbb{R}^{m_1 \times m_2 ... \times m_p}$. The samples belongs to one of c classes. An interpretable classification module should predict its class $y \in \{1, 2, .., c\}$ and give an explanation $\boldsymbol{I} \in \mathbb{R}^{c \times m_1 \times m_2 ... \times m_p}$. $\boldsymbol{I}_{j,i_1,i_2,...,i_p}$ represents the importance/impact/contribution of feature $\boldsymbol{X}_{i_1,i_2,...,i_p}$ in the context for classifying the sample to class j. ■

As can be seen from the definition of explanation, it provides the importance value of a feature not only for the predicted class, but also for other classes. In practice, the explanation does not have to be organized as a tensor $\boldsymbol{I}$. As long as an importance score of each element in $\boldsymbol{X}$ for each class can be computed, it is equivalent to having $\boldsymbol{I}$. The interpretability mentioned in this study refers to the intrinsic ability of classification models to provide the defined explanation.

Two simple models can achieve two of the aforementioned qualities except high classification performance. They are logistic regression and softmax regression, for binary classification and multi-class classification respectively. Any tensor $\boldsymbol{X} \in \mathbb{R}^{m_1 \times m_2 ... \times m_p}$ can be flattened to a vector $\boldsymbol{x} = flatten(\boldsymbol{X}) \in \mathbb{R}^m$, where $m = m_1 \times m_2 ... \times m_p$ is the number of features, by reorganizing the elements of a tensor to a 1d array to form a vector. Logistic regression can be expressed as:

$$y = \sigma(\boldsymbol{w}^T \boldsymbol{x} + b) = \sigma(w_1 x_1 + w_2 x_2 + ... + w_m x_m + b) \tag{3.1}$$

where $\boldsymbol{w} = (w_1, w_2, ..., w_m) \in \mathbb{R}^m, \boldsymbol{x} = (x_1, x_2, ..., x_m) \in \mathbb{R}^m$, $b \in \mathbb{R}$. Whether feature x_j makes the sample positive depends on the sign of $w_j x_j$. If $w_j x_j > 0$, x_j makes it positive, and vice versa. The degree of the impact of x_j on y depends on $|w_j x_j|$: a large $|w_j x_j|$ implies a large impact of x_j. If the model predicts a sample to be positive, the most influential factor that leads to the result is $max_j w_j x_j$. If the model predicts a sample to be negative, the most influential factor that leads to the result is $min_j w_j x_j$. In other words, the contribution of feature x_i to the positive class is calculated as $w_i x_i$, and the contribution of feature x_i to the negative class is $-w_i x_i$.

For multi-class classification, let c be the number of classes. Softmax regression can be expressed as follows:

$$\boldsymbol{y} = softmax(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}) \tag{3.2}$$

where $\boldsymbol{W} \in \mathbb{R}^{c \times m}$,$\boldsymbol{b} \in \mathbb{R}^{c}$, $softmax(\boldsymbol{z}) = \frac{1}{\sum_{j=1}^{c} e^{z_j}}(e^{z_1}, ..., e^{z_c})$. The output is a vector of dimension c, and each element is the probability that the sample belongs to a class. Therefore, $W_{i,j}x_j$ is the contribution of feature x_j to class i.

Thus, these two models can explain the classification results efficiently since only one forward running is required. The only thing that is missing is the classification performance. These two models do not allow features to interact with each other, so the coefficient for each feature, i.e., w_i and $W_{i,j}$, are static without considering the context.

To provide the same explanations that logistic regression and softmax regreesion can provide and the expressive ability of non-linear models, we propose a novel *interpretable feedforward neural network (IFFNN)* architecture. There are two versions of it, for binary classification and multi-class classification respectively. They can be seen as logistic regression and softmax regression with dynamically computed weights, rather than static weights. Thus, they have **the same interpretability as logistic regression and softmax regression**. They are described as follows.

For binary classification, the IFFNN can be expressed as follows:

$$\boldsymbol{v}(\boldsymbol{X}) = f(\boldsymbol{X}) \tag{3.3}$$

$$\boldsymbol{w}(\boldsymbol{X}) = \boldsymbol{W_2}\boldsymbol{v}(\boldsymbol{X}) + \boldsymbol{b_2} \tag{3.4}$$

$$\boldsymbol{x} = flatten(\boldsymbol{X}) \tag{3.5}$$

$$\boldsymbol{y} = \sigma(\boldsymbol{w}(\boldsymbol{X})^T\boldsymbol{x} + b) \tag{3.6}$$

where f represents an arbitrary feedforward neural network with any kinds of layers, $\boldsymbol{v}(\boldsymbol{X}) \in \mathbb{R}^{d}$, $\boldsymbol{W_2} \in \mathbb{R}^{m \times d}$, $\boldsymbol{b_2} \in \mathbb{R}^{m}$. The contribution of feature $X_{i_1,...,i_p}$ to the positive

class is $w(\boldsymbol{X})_i x_i$ where $i = (i_1 - 1) \times (m_2 m_3 ... m_p) + (i_2 - 1) \times (m_3 m_4 ... m_p) + ... + i_p$. The contribution of feature $X_{i_1,...,i_p}$ to the negative class is $-w(\boldsymbol{X})_i x_i$.

For multi-class classification, the IFFNN can be expressed as follows:

$$\boldsymbol{v}(\boldsymbol{X}) = f(\boldsymbol{X}) \tag{3.7}$$

$$\boldsymbol{W}(\boldsymbol{X}) = Reshape(\boldsymbol{W_2}\boldsymbol{v}(\boldsymbol{X}), (c \times m)) + \boldsymbol{B_2} \tag{3.8}$$

$$\boldsymbol{x} = flatten(\boldsymbol{X}) \tag{3.9}$$

$$\boldsymbol{y} = softmax(\boldsymbol{W}(\boldsymbol{X})\boldsymbol{x} + \boldsymbol{b}) \tag{3.10}$$

where $\boldsymbol{v}(\boldsymbol{X}) \in \mathbb{R}^d$, $\boldsymbol{W_2} \in \mathbb{R}^{(cm)\times d}$, $\boldsymbol{B_2} \in \mathbb{R}^{c\times m}$, $\boldsymbol{x} \in \mathbb{R}^m$, and $\boldsymbol{b} \in \mathbb{R}^c$. The contribution of feature $X_{i_1,...,i_p}$ to class j is $W(\boldsymbol{X})_{j,i} x_i$ where $(i_1-1)\times(m_2 m_3 ... m_p)+(i_2-1)\times(m_3 m_4 ... m_p)+ ... + i_p$.

It should be noted that assuming $\boldsymbol{v}(\boldsymbol{X})$, the output of $f(\boldsymbol{X})$, to be a vector of a fixed dimension does not cause the loss of generality. When $f(\boldsymbol{X})$ is a higher order tensor rather than a vector, its shape is still fixed, so it can always be converted to a vector by applying a $flatten$ operation.

As can be seen, the explanation of IFFNN is actually the byproduct of the classification result. With only one forward running, the classification result and the explanations are both acquired. Thus, the efficiency for the explanation is optimal.

3.1.1 Discussion

In some cases, in the input tensor, multiple elements correspond to the same object. When the contribution of each object is expected, the contributions of these elements should be added up. For instance, an RGB image can be represented as a third-order tensor $\boldsymbol{X} \in \mathbb{R}^{3\times h\times w}$. $X_{1,i,j}$, $X_{2,i,j}$, and $X_{3,i,j}$ are the red, green, and blue values of the same pixel. The contribution of pixel (i, j) is the summation of the contributions of $X_{1,i,j}$, $X_{2,i,j}$, and $X_{3,i,j}$.

3.2 Experiments

In this section, we evaluate various versions of IFFNNs on different datasets. The objectives are to answer the following questions:

- Is classification performance harmed when the feedforward neural networks are organized in our interpretable way compared to normal feedforward neural networks?
- Do the explanations given by the IFFNNs make sense?

3.2.1 Datasets

We evaluate the models on two datasets: MNIST [86] and INBEN. They complement each other in the evaluation procedure. MNIST is an image classification dataset that allows us to evaluate IFFNNs with convolutional layers and to qualitatively evaluate the interpretability of IFFNNs. However, it cannot be used to quantitatively evaluate their interpretability, since there is no exact answer on how important each pixel is for the classification results. With our created dataset INBEN, the gold standard explanations of the samples are known, and thus allows us to achieve this purpose.

Table 3.1: Statistics of the datasets used for evaluation.

Dataset	Training	Valid	Test	X Shape
MNIST 10 classes	50,000	10,000	10,000	(28,28)
MNIST 2 classes	10,554	2,111	2,115	(28,28)
INBEN 10 classes	100,000	10,000	10,000	(1000,)
INBEN 2 classes	20,000	2,000	2,000	(1000,)

MNIST

MNIST is a handwritten digit dataset. It is a common benchmark for image classification models. This dataset works well for our purposes because of its easily interpretable char-

acteristic. The IFFNNs applied on this dataset can point out which pixels are important to classify a sample to a certain digit. It is easy for humans to determine whether these pixels are good indicators for the predictions.

We create two scenarios with MNIST. **Scenario 1** uses samples of all 10 classes. In this scenario, we can evaluate the versions of IFFNNs for multi-class classification. **Scenario 2** uses samples of only two classes (digits of "0" and "1"). In this scenario, we can evaluate the versions of IFFNNs for both binary classification and multi-class classification.

INBEN

By visualizing the importance of each pixel of an image in MNIST, we can only qualitatively evaluate the interpretability of the IFFNNs. To quantitatively evaluate the interpretability, we propose a synthetic INterpretablility BENchmark (INBEN) dataset. It can be described as follows:

1. Each sample belongs to one of c classes.

2. Each sample is a vector of dimension m. Each entry corresponds to a fixed feature, and the value of it could be 0 or 1. For example, if $m = 5$, a sample could be (1 0 1 1 0).

3. For each class, there is a set of randomly generated patterns, where if a sample contains one of these patterns, it belongs to that class. For example, (1,3) is a pattern for class 2. It means that a sample x belongs to class 2 if $x_1 = 1$ and $x_3 = 1$. (1 0 1 1 0) is an example that contains this pattern.

4. There is a class priority sequence (e.g., [3,2,4,1,5]). If a sample contains patterns of multiple classes, it belongs to the class with the highest priority among them. For example, if a sample contains the patterns of both class 2 and class 5, it belongs to class 2.

5. There is a default class. If a sample contains no patterns, it belongs to the default class.

We also create two scenarios with INBEN datasets. **Scenario 1** contains samples of 10 classes, and **Scenario 2** contains samples of 2 classes.

The statistics of the datasets are given in Table 3.1.

3.2.2 Models

We include four kinds of feedforward neural networks in our experiments to illustrate the classification performance and interpretability of the IFFNN architecture. They are fully connected neural networks (FC), convolutional neural networks (CNN) [85], fully connected neural networks with highways (HW) [132], and residual neural networks (ResNET) [66]. For each of the four kinds of neural networks, we have eight different variants. We use FC as the example to describe the variants:

- **FC-BC1** A feedforward neural network with fully connected layers for binary classification. The top fully connected layer maps the feature vector to a real number followed by a sigmoid layer. This is only applicable to **Scenario 2**.
- **FC-MC1** A feedforward neural network with fully connected layers for multi-class classification. The top fully connected layer maps the feature vector to a vector of dimension c followed by a softmax layer.
- **FC-IFFNN-BC** The interpretable version of FC-BC1 achieved by replacing the top layer with Eq.3.4$\sim$3.6. This is only applicable to **Scenario 2**.
- **FC-IFFNN-MC** The interpretable version of FC-MC1 achieved by replacing the top layer with Eq.3.8$\sim$ 3.10.
- **FC-BC2** Similar to FC-BC1, with the total number of trainable parameters about the same as FC-IFFNN-BC by increasing the dimensions of the layers but not increasing the number of layers. This is only applicable to **Scenario 2**.

- **FC-MC2** Similar to FC-MC1, with the total number of trainable parameters about the same as FC-IFFNN-MC by increasing the dimensions of the layers but not increasing the number of layers.
- **FC-BC3** Similar to FC-BC1, with the total number of trainable parameters about the same as FC-IFFNN-BC by increasing the number of layers, and adjusting the dimension of each layer. This is only applicable to **Scenario 2**.
- **FC-MC3** Similar to FC-MC1, with the total number of trainable parameters about the same as FC-IFFNN-MC by increasing the number of layers, and adjusting the dimension of each layer.

For the other three kinds of neural networks, there are the same eight variants. When we apply FC and HW networks on the MNIST dataset, we flatten the input to a vector. We don't apply CNN and ResNET on INBEN because those two networks are mainly for input of matrices or third-order tensors.

We also compare with other interpretable models, including logistic regression (LR), softmax regression (SR), and decision trees (DT). We use grid search to tune the hyper-parameters of decision trees, including its split criterion and maximum depth. The candidate values are given in Table 3.2.

Table 3.2: Candidate values for hyper-parameters of decision tree.

Hyperparameter	Candidate Values
Split Criterion	gini,entropy
Maximum Depth	10,25,50,100,200,300,400,500,1000

3.2.3 Evaluation Metrics

We describe the evaluation metrics for classification performance and interpretability in this section.

For the classification performance, following the tradition, we use **accuracy** as the metric, which is the number of correctly classified samples over the total number of samples.

We cannot use MNIST to quantitatively evaluate the interpretability of the models, but we can use INBEN. With INBEN, we know the reason why a sample belongs to a class. It is the pattern(s) that decides its class. The ideal explanations should give the features included in the patterns the greatest contribution values. Therefore, we use the average of **accuracy@N** as our evaluation metric for interpretability. We formally define it as follows:

Definition 3.2.1 (Accuracy@N)**.** Let S_1 be the set of features in the pattern(s) that determines a sample $\boldsymbol{x}$ belong to class c. Let $N = |S_1|$. Let S_2 be the set of top N important features for classifying $\boldsymbol{x}$ to class c by an interpretable classification system. Let $S_3 = S_1 \cap S_2$ and $n = |S_3|$. Then, $Accuracy@N = n/N$.

As can be seen, N is variant to different samples. Below is an example.

A sample $\boldsymbol{x}$ belongs to class 2 because it contains the two patterns of class 2: (113,251) and (35,72,99,217,251). We thus have $S_1 = \{35, 72, 99, 113, 217, 251\}$ and $N = 6$. Let the top six most important features for classifying it to class 2 determined by an interpretable classification model be: 113,251,7,35,12,308. Then, we have $S_2 = \{7, 12, 35, 113, 251, 308\}$, $S_3 = \{35, 113, 251\}$ and thus $n = 3$. $Accuracy@N = \frac{3}{6} = 0.5$.

We use the average of accuracy@N over all correctly classified test samples as the evaluation metric for interpretability. We do not include wrongly classified samples because the $Accuracy@N$ of explanations for wrong predictions do not mean anything.

3.2.4 Experiment Setting

We train and evaluate the models on a server with two Xeon E5-2697 CPUs, 384 GB of memory, and four Nvidia Titan XP graphics cards. Only one graphics card is used for each run. The operating system is Windows Server 2016. We use Python 3.7.9 and PyTorch 1.6.0 [112] to implement the models. We use the implementation of DT in scikit-learn 0.23.2 [113].

Table 3.3: Classification performance evaluation on MNIST and INBEN.

Model	10-class MNIST		2-class MNIST		10-class INBEN		2-class INBEN	
	Params	Acc	Params	Acc	Params	Acc	Params	Acc
FC-MC1	898.5K	98.46	894.5K	99.93	1.0M	97.80	1.0M	98.23
FC-MC2	4.8M	98.54	1.7M	99.94	6.0M	98.83	2.0M	98.45
FC-MC3	4.8M	98.49	1.7M	99.92	6.0M	98.69	2.0M	98.37
FC-IFFNN-MC	4.8M	98.06	1.7M	99.91	6.0M	98.19	2.0M	99.06
HW-MC1	2.4M	98.13	2.4M	99.93	2.5M	97.99	2.5M	98.57
HW-MC2	6.3M	98.10	3.2M	99.92	7.5M	97.81	3.5M	98.69
HW-MC3	6.3M	97.67	3.2M	99.93	7.5M	97.41	3.5M	98.68
HW-IFFNN-MC	6.3M	97.96	3.2M	99.90	7.5M	97.58	3.5M	99.28
ResNET-MC1	226.2K	99.50	201.1K	99.92	NA	NA	NA	NA
ResNET-MC2	24.7M	99.41	5.1M	99.99	NA	NA	NA	NA
ResNET-MC3	24.7M	99.39	5.1M	99.93	NA	NA	NA	NA
ResNET-IFFNN-MC	24.8M	98.92	5.1M	99.95	NA	NA	NA	NA
CNN-MC1	1.2M	98.88	1.2M	99.89	NA	NA	NA	NA
CNN-MC2	72.3M	98.95	14.5M	99.92	NA	NA	NA	NA
CNN-MC3	72.3M	98.99	14.5M	99.93	NA	NA	NA	NA
CNN-IFFNN-MC	72.3M	98.69	14.5M	99.96	NA	NA	NA	NA
SR	7.8K	92.82	1.6K	99.95	10.0K	87.53	2.0K	97.67
DT	NA	88.19	NA	99.66	NA	76.75	NA	98.93
FC-BC1	NA	NA	894.0K	99.95	NA	NA	1.0M	98.04
FC-BC2	NA	NA	1.3M	99.92	NA	NA	1.5M	98.47
FC-BC3	NA	NA	1.3M	99.91	NA	NA	1.5M	98.58
FC-IFFNN-BC	NA	NA	1.3M	99.94	NA	NA	1.5M	98.67
HW-BC1	NA	NA	2.4M	99.92	NA	NA	2.5M	98.71
HW-BC2	NA	NA	2.8M	99.91	NA	NA	3.0M	98.55
HW-BC3	NA	NA	2.8M	99.92	NA	NA	3.0M	98.57
HW-IFFNN-BC	NA	NA	2.8M	99.94	NA	NA	3.0M	99.34
ResNET-BC1	NA	NA	197.9K	99.98	NA	NA	NA	NA
ResNET-BC2	NA	NA	2.7M	99.95	NA	NA	NA	NA
ResNET-BC3	NA	NA	2.7M	99.96	NA	NA	NA	NA
ResNET-IFFNN-BC	NA	NA	2.7M	99.91	NA	NA	NA	NA
CNN-BC1	NA	NA	1.2M	99.93	NA	NA	NA	NA
CNN-BC2	NA	NA	7.2M	99.93	NA	NA	NA	NA
CNN-BC3	NA	NA	7.2M	99.91	NA	NA	NA	NA
CNN-IFFNN-BC	NA	NA	7.2M	99.94	NA	NA	NA	NA
LR	NA	NA	0.8K	99.95	NA	NA	1.0K	97.66

We use Adam [76] with the initial learning rate $1e-3$ to train all the neural networks including LR and SR. The batch size is 256 and maximum epoch is 200. The accuracy on the test set at the epoch in which the accuracy on the validation set is the best is reported.

We repeat each group of experiments five times and report the average. We use random seeds from 0 to 4 for model initialization.

3.2.5 Classification Results

The classification performance of all models is shown in Table 3.3. The IFFNN version of different types of feedforward neural networks achieves slightly higher or lower accuracy compared with the non-interpretable ones in most cases (i.e., the difference is at most 1%). Between the same kind of neural networks with different amounts of trainable parameters, the difference in accuracy is minor as well. On datasets with 10 classes of samples, we can see a significant gap ($> 5\%$) between SR, DT, and the neural networks. This means that forming feedforward neural networks in the proposed interpretable way does not harm the classification performance and is as effective as a normal multi-layer feedforward neural network.

3.2.6 Interpretability Results

Table 3.4: Evaluation of interpretability with Accuracy@N on INBEN.

Model	10-class INBEN	2-class INBEN
SR	86.81	83.96
FC-IFFNN-MC	98.55	91.39
HW-IFFNN-MC	98.43	95.46
LR	NA	83.90
FC-IFFNN-BC	NA	90.58
HW-IFFNN-BC	NA	95.50

Quantitative Analysis

The Accuracy@N of LR, SR, and the IFFNNs on INBEN are reported in Table 3.4. As shown, the Accuracy@N of IFFNNs is always larger than 90%, which means when a sample is correctly classified, the IFFNNs can correctly point out the features in the patterns

that determine its class. This indicates that the explanations provided by them are accurate.

We can also see that the explanations given by IFFNNs are even more accurate than those given by LR and SR. The reason is that the INBEN dataset we created is non-linear, thus these linear models cannot always capture the patterns that determine the class of a sample. To be more specific, LR and SR can only model the relation between a feature and a class independently, however, the patterns require the models to be able to model the co-occurrences of different features. Multi-layer neural networks model interactions of different features through the computations in the hidden layers. This also reflects the fact that as multi-layer networks, the IFFNNs have the pattern recognition ability of non-linear models.

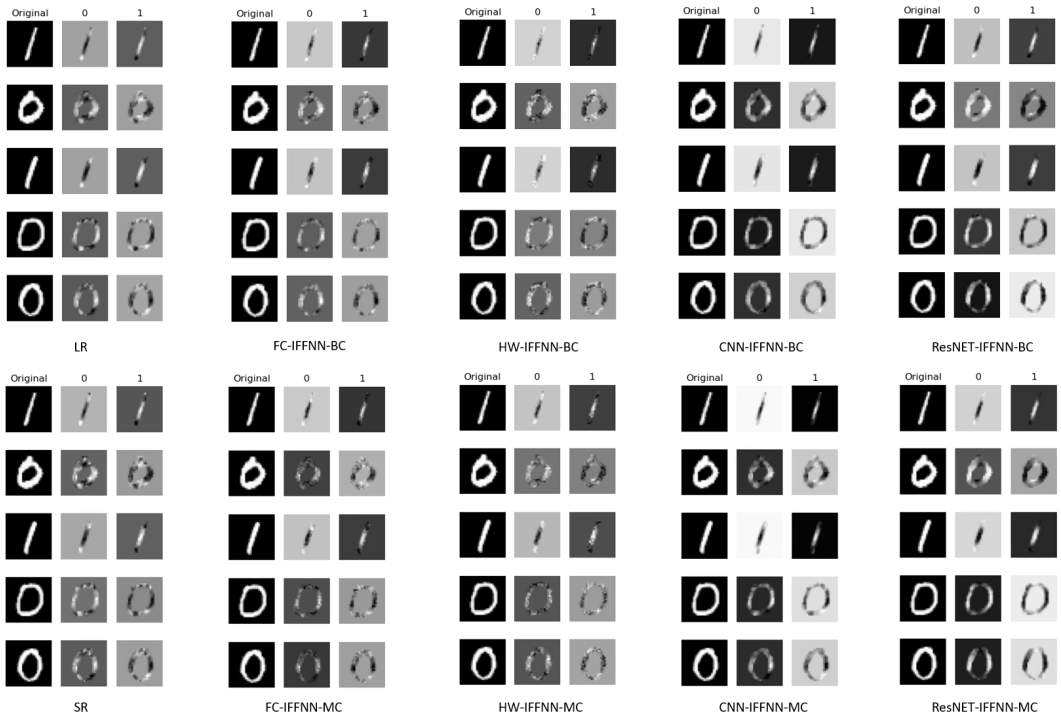

Figure 3.1: Examples of images and the explanations for the classifications on MNIST with only 0 and 1.

Qualitative Analysis

In addition to the quantitative evaluation, we also qualitatively evaluate the models on MNIST to manually check whether the explanations make sense. We show the importance of a pixel to a class in a greyscale image that has the same shape as the original image, and the greyscale of a pixel is the importance of the pixel in the same position. The greyscale of the background in the original images is always 0, so their importance is also 0. Therefore, the pixels that are lighter than the background provide a positive contribution to the class and the darker pixels provide a negative contribution. We use the scenario with only ”0” and ”1” for the evaluation because there are areas of the images that only contain white pixels for only one of them and these pixels are good indicators of the digits.

Figure 3.1 show some images from the test set and the importance images of them for all classes. We can see that for the images of ”0”, the important pixels for the right class (i.e., ”0”) determined by all IFFNNs focus on the pixels of the left and right parts of the circle. This makes sense because ”1” is usually close to a vertical bar, so its white pixels rarely appear in those areas of the images of ”1”. Therefore, it makes sense that white pixels appearing in these areas contribute more to the class of ”0”. The important pixels for images of ”1” are more concentrated in the center part of the stroke. This is also valid because there are rarely white pixels in the center areas of images of ”0”.

Chapter 4

Malware Detection

Different from signature-based malware detection methods that can only recognize known malware or some non-significant variants, machine learning-based malware detection methods [14, 42, 101, 126, 146] can automatically learn common patterns of malware from the feature space that have better generalization ability than manually crafted signatures. However, there are two major challenges for machine learning-based malware detection models.

Interpretability is one of the dominant features for classification models in some domains, such as healthcare and cybersecurity. In cybersecurity, the explanations can help malware analysts justify the classification results and create a knowledge base of malware samples. Linear models such as logistic/softmax regression and Naive Bayes produce interpretable results on vectorial data but usually yield inferior classification performance than non-linear models such as multi-layer feed-forward neural networks [26]. However, the hidden layers between the input and the logistic/softmax layer make multi-layer feed-forward neural networks lose the interpretability of logistic/softmax regression to directly attribute the impact of each feature. It's a challenge to keep interpretability as well as classification performance for feedforward neural networks.

As the payload of malware exists mainly in its assembly code, modelling the assembly code could provide important information for malware detection. However, it is chal-

Table 4.1: Sample result of our malware detection and its explanation, which includes the 5 factors that contribute most to the prediction and the most related assembly functions.

File: 05c199.exe		
Prediction: malicious		
Confidence: 100%		
Primary factors leading to the prediction of malicious		
Feature description	Feature value	Impact
Assembly code	N/A	14.56
Number of PE imports	8	5.12
Major operating system version	1	1.49
Frequency of the string "Sleep"	1	0.82
Frequency of the string ".data"	1	0.59
Most influential assembly functions		
sub_401010		
sub_4062AE		

lenging to model the whole assembly code of executables because they are very long sequences. An executable of 1 MB could have hundreds of thousands of instructions. Existing training approaches are not effective to train such long sequences, and the memory consumption for training such long sequences cannot be handled with standard hardware.

Deep learning models have achieved significant breakthroughs in understanding natural language when properly trained on large corpora [47, 115, 116]. *Transformer* [136] based models achieve state-of-the-art results in natural language understanding and generation [21, 47, 51, 115, 116, 118]. However, their successful applications are mainly on short text, i.e., sentence-level tasks such as paraphrase detection and sentiment analysis [47, 115], or on short-document texts such as reading comprehension and automatic summarization of news articles [51]. For example, the state-of-the-art sequence model *GPT-3* [21] can process sequences of a maximum length of 2,048 tokens. That makes the transference of the success of existing methods to understanding assembly code a challenge. Apart from the fact that assembly code is too long, the differences between natural

language and assembly code in the structure composition and basic units stand as another problem to solve.

Despite the fact that the assembly code of an executable is usually very long, it has an innate hierarchical structure: instructions form basic blocks, basic blocks form assembly functions, and assembly functions form the ensemble of assembly code (i.e., the full logic) of an executable. The lengths of basic blocks, assembly functions, and the ensemble of the assembly code of an executable in terms of their direct sub-units are usually within thousands. Based on this characteristic, we propose the *Galaxy Transformer* network. It contains three components, namely the *Satellite-Planet Transformer*, the *Planet-Star Transformer*, and the *Star-Galaxy Transformer*. They are three customized *Star-Plus Transformer* networks organized in a hierarchy in order to understand the semantic meaning of the assembly code of an executable at different levels: basic block, assembly function, and executable. The Star-Plus Transformer is our improved version of the *Star Transformer* [65], which was proposed for natural language understanding as a variant of the Transformer [136]. The time complexity and space complexity of the Transformer is $O(n^2)$, where n is the length of the token. The Star Transformer replaces the fully connected structure of the Transformer with a star-shaped topology to reduce the complexities to $O(n)$, and it achieves better results on modestly sized datasets. A comparison of the topology between the Transformer, the Star/Star-Plus Transformer, and the Galaxy Transformer is shown in Figure 4.1. Our proposed universe-like topology of the Galaxy Transformer makes it possible to train very long sequences.

To provide explanations for the detection results, we apply our novel *interpretable feed-forward neural network* (*IFFNN*) as the other key component of our full model, the *Interpretable MAlware Detector* (*I-MAD*). It has the modelling power of a multi-layer neural network and the interpretability of a logistic regression model. An example of the prediction and its explanation is given in Table 4.1. It shows the detection result of a target file, the confidence in the result, the primary contributing features that lead to the prediction, and the most related assembly functions.

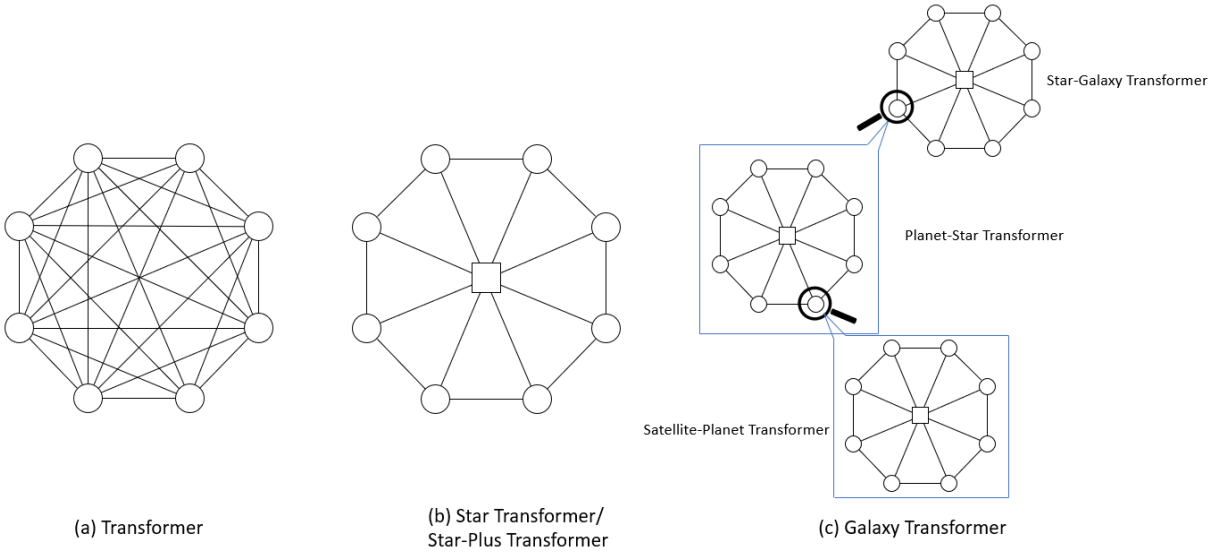

Figure 4.1: The comparison of topology of the Transformer, Star/Star-Plus Transformer, and Galaxy Transformer.

The contributions of the work described in this chapter are summarized below:

1. We propose the Galaxy Transformer as an early attempt in the literature to model the full sequences of assembly code for malware detection.

2. We propose two pre-training tasks to train the Satellite-Planet Transformer and Planet-Star Transformer, which are both components of the Galaxy Transformer, to understand the semantic meaning of assembly code at the basic block and assembly function levels.

3. We improve the way to use printable string features from previous works with our insights on malware.

4. We apply our novel IFFNN as the classification module of I-MAD to provide intrinsic explanations for detection results. The IFFNN module allows I-MAD to quantify the impact of each feature for the detection results.

The rest of the chapter is organized as follows. Section 4.1 defines the research problem. Section 4.2 describes the details of our proposed method. Section 4.3 presents the experiment results and analyses.

4.1 Problem Definition

In this section, we define some important concepts, followed by the definition of the research problem.

An executable is a sequence of bytes:

$$exe = \langle byte_1, byte_2, ... \rangle \tag{4.1}$$

The feature set of an executable is extracted by a set of extractors:

$$fea(exe) = \{ext_1(exe), ext_2(exe), ...\} \tag{4.2}$$

Except for assembly code, the other extracted features can be represented as a vector. We represent the assembly code as a series of nested sets and sequences.

The assembly code of an executable is a set of assembly functions:

$$code(exe) = \{f_1, f_2, ...\} \tag{4.3}$$

An *assembly function* is a set of basic blocks:

$$f = \{b_1, b_2, ...\} \tag{4.4}$$

A *basic block* is a sequence of assembly instructions:

$$b = \langle ins_1, ins_2, ... \rangle \tag{4.5}$$

An *assembly instruction* is a sequence of one opcode and two operands:

$$ins = \langle Opcode, Operand1, Operand2 \rangle \tag{4.6}$$

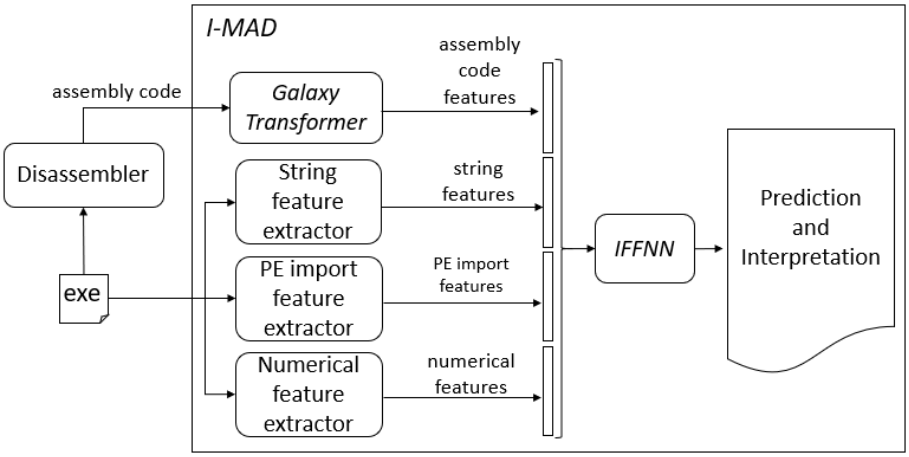

Figure 4.2: An overview of our I-MAD model.

For the uncommon instructions with three operands, the third is ignored. Empty operands are substituted by the special token `EMPTY`. All opcodes and operands form a set, and each of them is assigned an index number. Thus, one instruction can be abstracted as a sequence of three integers, where each integer represents an index of an opcode or operand.

Definition 4.1.1 (Malware Detection)**.** Consider a collection of executables E and a collection of labels L that show the executables in E are benign or malicious. Let exe be an unknown executable that $exe \notin E$. The *malware detection problem* is to build a classification model M based on E and L such that M can be used to determine whether the executable exe is benign or malicious. ■

4.2 Methodology

Our malware detection model I-MAD includes the Galaxy Transformer to learn a vector to represent the semantic meaning of the assembly code of an executable and our IFFNN that takes the vector representing the assembly code of a target executable and vectors representing other features as its inputs to produce an interpretable detection result. Figure 4.2 depicts an overview of our malware detection model. In this section, we introduce

the Star Transformer and describe how we improve it to form the Star-Plus Transformer to build the Galaxy Transformer. Then, we propose two methods to pre-train different components of the Galaxy Transformer. Next, we introduce the other features we use and how we apply the IFFNN to explain the detection results.

4.2.1 Galaxy Transformer

The Galaxy Transformer is proposed to learn vector representations for the semantic meanings of assembly code at the basic block, assembly function, and executable levels. The semantic meaning refers to the overall purpose (i.e., function) of assembly code.

Star Transformer

The Star Transformer [65] adopts the multi-head attention from the standard Transformer:

$$
\begin{aligned}
MultiAtt(q, H) &= Concat(head_1, ..., head_h)W^O \\
where\ head_i &= Attention(qW_i^Q, HW_i^K, HW_i^V) \\
Attention(q_i, K_i, V_i) &= softmax(\frac{q_i K_i^T}{\sqrt{d_{model}}})V_i
\end{aligned}
$$

where $K_i = HW_i^K$,$V_i = HW_i^V$, $W_i^Q \in R^{d_{model} \times d_k}, W_i^K \in R^{d_{model} \times d_k}, W_i^V \in R^{d_{model} \times d_v}, W^O \in R^{hd_v \times d_{model}}$ are learnable parameters, $q \in R^{d_k}$ is a query vector, and $H \in R^{n \times d_k}$ is a matrix that contains vector representations of n items to attend to. To compute the self-attention of a sequence $X = \langle x_1, x_2, ..., x_n \rangle$, in the Transformer, each x_i is a query q, and it attends to all items in the sequence, so $H = X$. Thus, its computational complexity is $O(n^2)$.

To reduce the computational complexity, the Star Transformer only considers connections between adjacent items and between a relay node and each item, as shown in Figure 4.1b. First, for each item x_i, a vector e_i is computed as the summation of its non-contextual semantic embedding and its positional encoding in the same way as the Transformer

does:

$$e_i = Emb(x_i) = SE(x_i) + PE(i)$$
$$E = [e_1; ...; e_n]$$

Then, the embeddings are fed into a multi-layer neural network to compute the hidden state for each x_i. h_i^t represents the hidden state of x_i at layer t. h_i^0 is initialized as e_i. The initial hidden state of the additional relay node is $s^0 = \frac{1}{n}\sum_n e_i$. To compute h_i^t, its context matrix C_i^t is formed by the hidden states of itself h_i^{t-1} and its adjacent nodes of the previous layer $h_{i-1}^{t-1}; h_i^{t-1}$, its embedding e_i, and the hidden state of the relay node s^{t-1}:

$$C_i^t = [h_{i-1}^{t-1}; h_i^{t-1}; h_{i+1}^{t-1}; e_i; s^{t-1}]$$

So, we have $C_i^t \in R^{6d_{model}}$.

At each layer, we have

$$h_i^t = LayerNorm(ReLU(MultiAtt(h_i^{t-1}, C_i^t)))$$
$$H^t = [h_1^t; ...; h_n^t]$$
$$s^t = LayerNorm(ReLU(MultiAtt(s^{t-1}, H^t)))$$

Thus, the relay node s^t serves as a global information collector. h_i^t collects local information from its adjacency nodes and global information from s^{t-1}. The computational complexity to compute all h_i^t is $O(n)$, and to compute s^t it is also $O(n)$. The overall computational complexity is therefore $O(n)$.

To put it all together, we represent a Star Transformer Layer as follows:

$$H^{t+1}, s^{t+1} = STL^t(H^t, s^t, E)$$

The full computation of the Star Transformer is as follows:

$$E = [Emb(x_1); ...; Emb(x_n)]$$

$$H^0 = E, s^0 = \frac{1}{n}\sum_n e_i$$

$$H^T, s^T = STL^T(STL^{T-1}(...STL^1(H^0, s^0, E), E), E)$$

Star-Plus Transformer

As previously shown, the Star Transformer can generate a contextual vector representation for each item in a sequence and a vector representation for the whole sequence with $O(n)$ computational complexity. We propose the following modifications for better performance.

1. There is no obvious reason why e_i should be in the context matrix C_i^t, so we remove e_i from C_i^t, resulting in $C_i^t = [h_{i-1}^{t-1}; h_i^{t-1}; h_{i+1}^{t-1}; s^{t-1}]$.

2. There was a pointwise feedforward neural network ($FNN = max(0; xW_1 + b_1)W_2 + b_2$) after the multi-head attention computation in the Transformer, but it is removed in the Star Transformer without an explanation. We add it back to compose the information collected by all attention heads and to generate higher-level features for the next layer.

3. A max-pooling on H^T across the top layer mixed with s^T was used as the representation for the whole sequence in the Star Transformer. We use only s^T to represent the whole sequence, since it has collected global information of the sequence.

To put it together, we have a Star-Plus Transformer layer $H^{t+1}, s^{t+1} = SPTL^t(H^t, s^t)$ computed as follows:

$$
\begin{aligned}
h_i^{t\prime} &= LayerNorm(ReLU(MultiAtt(h_i^{t-1}, C_i^t))) \\
h_i^t &= LayerNorm(ReLU(FFN(h_i^{t\prime}))) \\
H^t &= [h_1^t; ...; h_n^t] \\
s^{t\prime} &= LayerNorm(ReLU(MultiAtt(s^{t-1}, H^t))) \\
s^t &= LayerNorm(ReLU(FFN(s^{t\prime})))
\end{aligned}
$$

4.2.2 Satellite-Planet Transformer to Understand Basic Blocks

As we have stated before, a basic block is a sequence of assembly instructions: $b = \langle ins_1, ins_2, ... \rangle$. The objective of the Satellite-Planet Transformer is to learn a vector representation for b using its instructions. To build the Satellite-Planet Transformer with the Star-Plus Transformer, we modify the input layer of the latter because each instruction is not an atomic item, but a sequence of three items (i.e., an opcode and two operands). Since both the embedding of an instruction ins_i and its positional encoding should have d_{model} dimensions, we make the embeddings of the opcode and operands $d_{model}/3$ dimensions and use the concatenation of them as the embedding of the instruction. It is then added with the positional encoding to form e_i. The concatenation of the vector representation of the opcode and operands to form the vector of an instruction was also previously adopted by Ding et al. [50]. For the output, we directly use s^T, which is the representation of the relay node at the top layer as the semantic meaning representation of the basic block. To train the Satellite-Planet Transformer we propose the *Masked Assembly Model* task.

Definition 4.2.1 (Masked Assembly Model)**.** Let (b, ins) be a *basic block and assembly instruction pair*. Consider a set of basic block and assembly instruction pairs B. For each pair $(b, ins) \in B$, there is one mask instruction m in b that should originally be ins. Let t be a target basic block that is not in any pair of B, and one of its instructions is replaced

by m. The *Masked Assembly Model* task is to build a classification model M based on B to predict the original instruction t replaced by m. ■

This task is inspired by the *Masked Language Model* task proposed by Devlin et al. [47]. In that task, the authors mask random words from sentences and use the Transformer to predict the masked words based on the contextual words in the sentences. Their method is to feed the output vector of the Transformer corresponding to a masked word to an output softmax over the vocabulary. The prediction requires both global context and local context. The global context means the semantic meaning of the whole sentence except the masked word. The local context means the position of the masked word and its surrounding words that could indicate what ingredient the missing word should be. As the output vector corresponding to the masked word is the only information source for the output layer to make the prediction, it has to capture both global and local context. This does not fit our objective, since the output vector should only contain the semantic meaning of a basic block (i.e., global contextual information). Therefore, we separate the two kinds of information in two vectors: s^T containing the global contextual information and the output vector of the masked instruction $m = [MASK\_OPC, EMPTY, EMPTY]$ containing the local contextual information. We concatenate these two vectors to form one vector and feed it to three feed-forward neural networks with softmax over the whole set of opcodes and operands to predict the opcode and two operands of the original masked instruction. It should be noted that after this training step, we only need to keep the Satellite-Planet Transformer, which generates s^T, the semantic representation of the entire basic block, because the three feed-forward neural networks to predict the original masked instruction are not needed after the training for the *Masked Assembly Model* task.

4.2.3 Planet-Star Transformer to Understand Assembly Function

The Planet-Star Transformer is another customized Star-Plus Transformer built on top of the Satellite-Planet Transformer to learn the vector representation of the semantic meaning of an assembly function f from the set of vectors representing its basic block $\{b_1, b_2, ...\}$.

As the input is already vectors rather than integers, we abandon the input embedding layer of the Star-Plus Transformer that maps integers to embeddings. We directly feed the vectors representing the basic blocks in positional order to form a sequence to the Planet-Star Transformer, which is a Star-Plus Transformer without an input layer. We use s^T as the vector representation of the assembly function. To train the Planet-Star Transformer, we propose the *Assembly Function Clone Detection* task.

Definition 4.2.2 (Assembly Function Clone Detection)**.** Let (f_1, f_2) be an *assembly function pair*. Let (f_1, f_2, l) be a *labeled assembly function pair* in which the label l indicates whether the two assembly functions f_1 and f_2 are clones (i.e., semantically equivalent) of each other. Consider a collection of labeled assembly function pairs F. Let $p = (f_1, f_2)$ be a new function pair that p is not any function pair in F. The *assembly function clone detection task* is to build a classification model M based on F to determine whether the two functions in p are clones of each other. ■

The intuition is that if the vector representations of assembly functions can be used to determine whether two functions are clones of each other, then they contain the semantic meaning of the assembly functions. We train the network to generate similar vectors in cosine measure (i.e., $cos(s^T_{f1}, s^T_{f2})$) for real assembly function clone pairs and dissimilar vectors for non-clone pairs. The way we form the function pair dataset is described in Section 4.3.

4.2.4 Star-Galaxy Transformer to Understand Full Logic of Executable

Next, we use the Star-Galaxy Transformer to learn one vector representing the full logic of an executable based on the representations of all its assembly functions: $\{f_1, f_2, ...\}$. Technically, this is similar to learning the representation of an assembly function from the representations of its basic blocks, since both are intended to learn one vector representation from a set of vectors. Therefore, the Star-Galaxy Transformer is a duplicate of the Planet-Star Transformer. Their difference is that they work at different levels of the

hierarchy. The representation of the assembly code of an executable generated by the Star-Galaxy Transformer is fed to the IFFNN for malware detection without other pre-training tasks proposed for it.

With this, we have completely described how we build the Galaxy Transformer with three customized *Star-Plus Transformers* in a hierarchy to compute the vector representation of the assembly code of an executable.

4.2.5 Other Features

When malware is packed, or is polymorphic or metamorphic, the assembly code of its payload is encrypted and not statically accessible. Hence, using only assembly code would fail to identify its malicious purpose. According to the experience of previous works [8, 72], static analysis can still be effective, because the use of the stealthy mechanisms can be captured when analyzed from multiple static feature scopes. Next, we describe the three kinds of features we use and how we improve the way to use them.

Printable Strings

According to the literature [41, 69, 72, 127], printable strings are important features, because they include, among others, runtime-linked libraries, functions, and registry keys that are commonly used by malware, system paths, and sometimes the names of user-defined functions. Hence, we extract printable strings from the whole byte sequence of an executable. In our algorithm, a continuous subsequence is a *printable string* if it satisfies three conditions: 1) all of its bytes are ASCII characters, 2) it is terminated with a null symbol, and 3) its length is at least 5 bytes. We count the number of instances of each printable string in the training set and put the strings that appear more than a certain threshold, 1,000 in our case, in the frequent string set. Their frequencies in an executable are used as features. This is not new compared to previous works. The improvement is that we also use the number of printable strings that are not in the frequent string set, i.e., uncommon strings, as a feature, and we use the total number of common printable

Table 4.2: PE header numerical fields we use.

Machine	SizeOfOptionalHeader
Characteristics	MajorLinkerVersion
MinorLinkerVersion	SizeOfCode
SizeOfInitializedData	SizeOfUninitializedData
AddressOfEntryPoint	BaseOfCode
BaseOfData	ImageBase
SectionAlignment	FileAlignment
MajorOperatingSystemVersion	MinorOperatingSystemVersion
MajorImageVersion	MinorImageVersion
MajorSubsystemVersion	MinorSubsystemVersion
SizeOfImage	SizeOfHeaders
CheckSum	Subsystem
DllCharacteristics	SizeOfStackReserve
SizeOfStackCommit	SizeOfHeapReserve
SizeOfHeapCommit	LoaderFlags
NumberOfRvaAndSizes	SectionsNb
SectionsMeanEntropy	SectionsMinEntropy
SectionsMaxEntropy	SectionsMeanRawsize
SectionsMinRawsize	SectionsMaxRawsize
SectionsMeanVirtualsize	SectionsMinVirtualsize
SectionMaxVirtualsize	ImportsNbDLL
ImportsNb	ImportsNbOrdinal
ExportNb	ResourcesNb
ResourcesMeanEntropy	ResourcesMinEntropy
ResourcesMaxEntropy	ResourcesMeanSize
ResourcesMinSize	ResourcesMaxSize
LoadConfigurationSize	VersionInformationSize

strings in the executable as another feature. This is based on the intuition that encrypted malware has more uncommon printable strings and benign software has more common strings.

PE Imports

PE Imports are dynamically linked libraries and functions shown in the import address table of PE headers. The imports of an executable often illustrate its behaviors, e.g., mod-

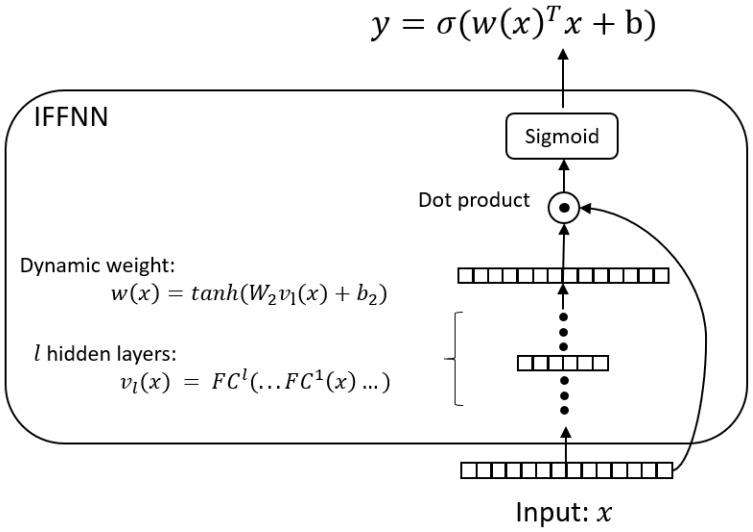

Figure 4.3: The architecture of the IFFNN applied in I-MAD.

ify the registry or hook a procedure [126, 127]. Therefore, we use the imports of dlls and their functions as another group of features.

PE Header Numerical Features

There are many numerical fields in PE headers that contain information that could form different patterns among malware and benignware [14, 126]. Hence, we also use these values as features. The fields of PE headers we use are given in Table 4.2.

We concatenate the vector representing the full logic of an executable v_{code}, printable string feature vector v_{str}, PE header numerical feature vector v_{num}, and PE import feature vector v_{imp} to form a vector representation of the executable from multiple scopes $v = [v_{code}, v_{str}, v_{num}, v_{imp}]$.

4.2.6 Interpretable Feed-Forward Neural Network

To tell the contribution of each feature for the detection result, we apply our proposed IFFNN as the classification module. Figure 4.3 illustrates its architecture.

Let $x \in \mathbb{R}^m$ be a feature vector representing a sample. We first feed it to l fully-connected hidden layers:

$$v_l(x) = FC^l(...FC^1(x)...) \tag{4.7}$$

$$where\ FC^i(v_{i-1}(x)) = tanh(W_1^i v_{i-1}(x) + b_1^i) \tag{4.8}$$

where $W_1^i \in \mathbb{R}^{d^i \times d^{i-1}}$, $b_1^i \in \mathbb{R}^{d^i}$, and $v_l(x) \in \mathbb{R}^{d^l}$. Then, we apply another normal fully-connected layer of which the output vector has the same dimension as x:

$$w(x) = W_2 v_l(x) + b_2 \tag{4.9}$$

where $W_2 \in \mathbb{R}^{m \times d^l}$, $b_2 \in \mathbb{R}^m$, and $w(x) \in \mathbb{R}^m$. $w(x)$ serves as a weight vector for each feature in x. The final confidence that the input sample is positive (in malware detection, positive means malicious) is calculated as follows:

$$y = IFFNN(x) = \sigma(w(x)^T x + b) \tag{4.10}$$

$$where\ \sigma(z) = \frac{1}{1 + e^{-z}} \tag{4.11}$$

where $b \in \mathbb{R}$ is a bias term. This is similar to a logistic regression (i.e., $y = \sigma(w^T x + b)$, where w is a parameter vector), and the difference is that our weight vector $w(x)$ is dynamically computed based on x rather than static parameters.

We feed v, the feature vector from multiple scopes of an executable, to the IFFNN to get the confidence y that it is malicious: $y = IFFNN(v) = \sigma(w(v)^T v) = \sigma(w(v)_1 v_1 + w(v)_2 v_2 + ... + w(v)_m v_m)$ and explain the result. If $|w(v)_j v_j|$ is large and $w(v)_j v_j > 0$, feature v_j has a large impact on the prediction of malicious. If $|w(v)_j v_j|$ is large and $w(v)_j v_j < 0$, feature v_j has a large impact on the prediction of benign. For printable string features, PE imports, and PE header numerical features, each dimension of their vector representations corresponds to a specific feature. The features can be the frequency of a certain string, whether

a certain DLL is imported, the value of a certain numerical field, etc. By checking its $w(v)_j v_j$, we know whether it makes the executable more likely malicious or benign. For the vector representing the full logic of an executable: v_{code}, each of its dimensions has no specific meaning, but we can see the impact of the full logic of the executable by computing the summation of the impact of each dimension of its vector: $\sum_{j \in v_{code}} w_{code,j} v_{code,j}$. If it is positive, from the perspective of the assembly code, the executable is more likely malicious, and vice versa.

As v_{code} is computed by our Star-Galaxy Transformer network, the attention weights of the assembly functions to the relay node at the top layer indicate the importance of each assembly function. We compute the summed attention weights of each assembly function over all heads to the relay node to determine which assembly functions are the main factors that influence the detection results.

4.2.7 Model Training

To train the Satellite-Planet Transformer, the objective function is the cross entropy loss of the prediction on the masked opcode and operands against the real opcode and operands. To train the Planet-Star Transformer and simultaneously fine-tune the Satellite-Planet Transformer, the objective function is the mean squared error between the computed cosine similarity between two assembly functions and the gold standard (i.e., 1 for clone function pairs, and -1 for non-clone function pairs). To train the full top-level network including the IFFNN and the Star-Galaxy Transformer, the objective function is the cross entropy loss of the prediction against the real label. To ensure that the Star-Galaxy Transformer gets sufficient training, we first train it without concatenating any other feature, i.e., feed v_{code} instead of v to the IFFNN ($y = IFFNN(v_{code})$), and train it for malware detection. This is in fact the pre-training of the Star-Galaxy Transformer. Then, we concatenate v_{code} with other features to feed it to the IFFNN ($y = IFFNN(v)$), and train it the same way for malware detection. The Satellite-Planet Transformer and Planet-Star Transformer networks are not fine-tuned when we train the top-level network. For all the

training objectives, we use Adam [76] with the initial learning rate $1e-4$. We use early stopping with the validation set to avoid overfitting [24].

4.3 Experiments

The objectives of our experiments are to 1) evaluate the performance of I-MAD for malware detection, 2) compare I-MAD to other state-of-the-art static malware detection solutions, and 3) demonstrate the interpretability of I-MAD.

We train and evaluate the models on a server with two Xeon E5-2697 CPUs, 384 GB of memory, and four Nvidia Titan XP graphics cards. We use PyTorch [112] to implement our model. We use the "pefile"[1] library to extract numerical features from PE headers.

4.3.1 Datasets and Pre-training

Table 4.3: Top 10 majority malware families of the dataset.

Malware Family	Number	Percentage
Fareit	9,436	8.2%
Zbot	6,433	5.6%
Emotet	6,343	5.5%
Gandcrab	4,120	3.6%
Mepaow	4,055	3.5%
CobaltStrike	3,151	2.7%
Allaple	2,081	1.8%
Ursnif	1,552	1.3%
Autoit	1,017	1.0%
NaKocTb	794	0.7%
Total	38,982	33.9%

For the two pre-training tasks, we compile several open source projects that are compatible with GCC and/or LLVM. We choose these two compilers because they are the

[1]https://github.com/erocarrera/pefile

most appropriate options to provide different compilation options to generate semantically equal but literally different assembly functions. GCC compiler provides four different optimization levels (i.e., O0, O1, O2, and O3) to compile projects. We compile *busybox*, *coreutils*, *libcurl*, *libgmp*, *libtomcrypt*, *libz*, *magick*, *openssl*, *puttygen*, and *sqlite3* with GCC at all four optimization levels. Thus, for every assembly function in those projects we have four semantically equivalent versions. O-LLVM[2] is an obfuscator of the LLVM compiler that provides control flow flattening, instruction substitution, and bogus control flow obfuscation mechanisms. We use O-LLVM to compile *libcrypto*, *libgmp*, *libMagickCore*, and *libtomcrypt* with five different settings: no obfuscation, each of the three obfuscation mechanisms, and all three mechanisms. Thus, we have five versions of their every function. We use IDA Pro[3], a commercial disassembler, to disassemble our compiled executables and acquire the assembly functions.

We use basic blocks of lengths between 5 to 250 instructions to form our *Masked Assembly Model* dataset; these blocks are within the typical length range of blocks that provide enough context and are not too long to harm training efficiency. As a result, this dataset contains 38,427,440 basic blocks. We use all of them for training and none for testing as the purpose of the dataset is to train the Satellite-Planet Transformer to understand assembly code, and the accuracy of this task is uninformative.

We use the semantically equivalent but literally different functions we compiled to form real function clone pairs. We randomly pair the same number of functions to be non-clone function pairs to create the dataset for the Assembly Function Clone Detection task. We limit the maximum number of instructions per basic block to be 50 and the maximum number of blocks per function to be 50 in this dataset, so that the memory of our graphics cards can hold the data flowing in the bottom two-level networks. There are 213,656 function pairs in the training set, 26,898 functions in the validation set, and 26,746 functions in the test set. Our bottom two-level networks get a classification accuracy of 91.5% on the test set. This means that the assembly function representations it

[2]https://github.com/obfuscator-llvm/obfuscator/wiki
[3]https://www.hex-rays.com/products/ida/

computes and the representations of basic blocks that are fed to it indeed capture the semantic meanings of assembly code. We do not elaborate on the experiments for this task since it is not the objective task, but rather a task to pre-train the Planet-Star Transformer and fine-tune the Satellite-Planet Transformer.

For malware detection we collected a dataset containing 115,000 benign and 115,000 malicious executables. There is no redundancy in the dataset. Following the literature [78, 117, 127], the benign executables are the .exe and .dll files from the installation paths of software programs. The malicious executables are collected from *MalShare* and *VirusShare*. The top 10 major malware families of the dataset are presented in Table 4.3. They are obtained with ClamAV [4]. The top 10 known packers that are applied on the malware samples are shown in Table 4.4. The usage of packers is acquired with Yara Rules [5]. The way we split the dataset into training set, validation set, and test set is introduced in Subsection 4.3.3.

Table 4.4: Top 10 packers used in the malware dataset.

Packer	Number	Percentage
UPX	7,776	6.7%
BobSoft Mini Delphi	5,262	4.5%
ASProtect	1,826	1.59%
ASPack	1,780	1.55%
PECompact	586	0.51%
Armadillo	369	0.32%
D1S1G	155	0.14%
WinrarSFX	124	0.11%
MoleBox	69	0.06%
WinZipSFX	38	0.03%
Total	17,985	15.6%

4.3.2 Models for Comparison

We compare our *I-MAD* model to several state-of-the-art static malware detection models.

[4] https://www.clamav.net/
[5] https://github.com/Yara-Rules/rules

Table 4.5: Results of k-fold cross-validation experiment. It includes the p-values (pv) of t-test for F1 and accuracy between *I-MAD* (ST+) and other models.

Model	P	R	F1	pv (F1)	Acc	pv (Acc)
Mosk2008OB	96.1	95.8	95.9	3.3e-13	95.9	1.6e-20
Bald2013Meta	96.5	95.9	***96.2***	1.1e-13	***96.2***	6.7e-20
Saxe2015Deep	95.2	96.1	95.7	4.0e-14	95.6	4.5e-21
Raff2017MalC	95.9	96.3	96.1	5.6e-15	96.1	4.0e-20
Krcal2018Conv	93.2	93.2	93.2	1.7e-15	93.2	1.0e-23
Mour2019CNN	72.6	71.5	72.0	2.3e-26	71.8	1.5e-30
SVM (same features)	96.1	96.4	***96.2***	3.7e-13	***96.2***	5.6e-20
I-MAD (no code)	96.5	96.6	96.5	9.8e-13	96.5	4.0e-19
I-MAD (ST)	97.0	97.9	97.3	5.0e-3	97.2	4.7e-6
I-MAD (ST+)	97.5	97.9	**97.7**	N/A	**97.7**	N/A

- **Mosk2008OB** Moskovitch et al. [101] propose to use TF or TF-IDF of opcode bigrams as features and use document frequency (DF), information gain ratio, or Fisher score as the criteria for feature selection. They apply Artificial Neural Networks, Decision Trees, Naïve Bayes, Boosted Decision Trees, and Boosted Naïve Bayes as their malware detection models.

- **Bald2013Meta** Baldangombo et al. [14] propose to extract multiple raw features from PE headers and use information gain and calling frequencies for feature selection and PCA for dimension reduction. They apply SVM, J48, and Naïve Bayes as their malware detection models.

- **Saxe2015Deep** Saxe et al. [126] propose a sophisticated deep learning model that works on four different features: byte/entropy histogram features, PE import features, string 2D histogram features, and PE metadata numerical features. We tried to follow the exact features they extract when we implement it, but they do not provide the exact metadata numerical fields they use, so we just use the same numerical fields of PE headers used in our model as part of their input.

- **Raff2017MalC** Raff et al. [117] treat an executable as a sequence of bytes and apply a gated 1D convolutional neural network (CNN) to classify an executable. The net-

work includes an embedding layer, two convolutional layers with large filters and strides, a global max-pooling layer, and two fully-connected layers. The output of one convolutional layer serves as the gate of the other.

- **Krcal2018Conv** Following Raff et al. [117], Krcal et al. [79] treat an executable as a sequence of bytes and apply a CNN for malware detection, but their CNN is deeper and has smaller filters. There are four convolutional layers and four fully connected layers. Instead of a global max-pooling layer, they use a global mean-pooling layer after the convolutional layers.

- **Mour2019CNN** Mourtaji et al. [102] convert malware binaries to grayscale images and apply a 2D CNN on malware images for malware classification.

For the papers in which the authors describe multiple ways to select features and/or apply multiple machine learning models ([14, 101]), we try with all possible settings and report the best results that their methods can achieve to compare with our model.

As the ablation study, we also compare our full model "*I-MAD* (ST+)" with "*I-MAD* (no code)" and "*I-MAD* (ST)". "*I-MAD* (no code)" is our model without using assembly code. These comparisons can show the effectiveness of modeling assembly code with Galaxy Transformer. "*I-MAD* (ST)" is to build the Galaxy Transformer with the original Star Transformer, rather than the Star-Plus Transformer, to show the effectiveness of our modifications.

We also compare our model with an SVM model that uses the same features as *I-MAD* except for assembly code, since it is not a vectorial feature. We consider linear, polynomial, and RBF kernels and use grid search for tuning hyper-parameters. Comparing this baseline with *I-MAD* (no code), we can separately show the effectiveness of the feature set and our model.

4.3.3 Experiment Settings

We evaluate the models under two different experiment settings. The main evaluation metric is accuracy (Acc), but we also evaluate the models with precision (P), recall (R), and F1.

- **K-Fold Cross-Validation** We first evaluate our model and others with k-fold cross-validation where $k = 5$. The original dataset is randomly split into 5 even subsets. Each subset takes a turn to be chosen as the test set. Another subset takes a turn to be chosen as the validation set. The other 3 subsets form the training set. Thus, we have ${}^5P_2 = 20$ different experiment groups. Each group contains 138,000 samples in the training set, 46,000 in the validation set, and 46,000 in the test set. We acquire the experiment results of the 20 groups and report the averages.

- **Time Split Evaluation** In addition to cross-validation evaluation, we also evaluate the models in a more challenging and realistic scenario. In real life, a malware detection system is expected to detect new malware with its knowledge of known malware. To evaluate this ability of the models, we follow Saxe et al. [126] to perform a time split experiment. We use the executables compiled before 2015 to form the training and validation set, and those compiled after 2017 to form the test set. We exclude samples with a compilation time before 2000 or after 2020, either because the compilation dates are fake or the samples are outdated. There are 106,000 samples in the training set, 20,000 in the validation set, and 40,000 in the test set. We run each model with different initialization and random seeds 5 times and report the averages of the results.

4.3.4 Results

The results of the k-fold cross-validation and the time split experiments are shown in Table 4.5 and Table 4.6, respectively.

Table 4.6: Results of time split experiment. It includes the p-values of t-test for F1 and accuracy between *I-MAD* (ST+) and other models.

Model	P	R	F1	pv (F1)	Acc	pv (Acc)
Mosk2008OB	88.6	88.6	88.6	1.2e-15	88.6	3.4e-22
Bald2013Meta	88.3	88.1	88.2	1.6e-17	88.2	1.2e-22
Saxe2015Deep	87.4	87.7	87.5	1.4e-17	87.5	2.6e-23
Raff2017MalC	88.5	89.0	***88.7***	1.2e-16	***88.7***	4.5e-22
Krcal2018Conv	84.2	83.2	83.7	3.1e-18	83.8	1.2e-27
Mour2019CNN	57.0	56.6	56.8	4.3e-31	56.9	7.1e-34
SVM (same features)	89.2	88.8	***89.0***	7.3e-15	***89.0***	6.1e-22
I-MAD (no code)	89.4	89.6	89.5	3.2e-15	89.5	6.7e-21
I-MAD (ST)	91.1	91.1	91.1	8.6e-11	91.1	2.6e-15
I-MAD (ST+)	91.4	91.6	**91.5**	N/A	**91.5**	N/A

The full version of *I-MAD* achieves statistically significantly better accuracy and F1 than the other models in all experiments, as the p-values in t-test are much smaller than 0.01. The improvements of our model on accuracy and F1 are larger in the time split experiments than in the cross-validation experiment. Even though we make sure there is no redundancy in the dataset, some pieces of malware could be extensively similar to each other if they are from the same family and compiled with slightly different modifications. Also, their compilation time is usually close to each other. In the time split experiment, the executables in the test set are compiled at least 2 years later than any executable in the training and validation sets. This is a more difficult setting that can be reflected in the pervasively lower accuracy in the time split setting than in the cross-validation setting. Thus, the significantly larger improvement of our detection model over other models in the time split experiment indicates that it has better abilities to learn robust and consistent patterns from old samples that can be generalized to classify new samples.

It is clear that with modelling assembly code with the Galaxy Transformer, *I-MAD* achieves much better results than it does without modelling the assembly code. This shows that modelling assembly code with our Galaxy Transformer helps in differentiating malicious and benign executables. We can also see that the Galaxy Transformer built with

Star-Plus Transformer (*I-MAD* (ST+)) is more effective than the one built with the original Star Transformer (*I-MAD* (ST)). This confirms that our modifications are useful.

SVM with the same features as *I-MAD* except for assembly code, achieves accuracy similar to other best baseline models in the cross-validation experiment, and it achieves better accuracy than other baseline methods in the time split experiment, while worse than *I-MAD* (no code). This shows that the feature set we propose is effective, and our IFFNN has advantages in classification performance on the same feature set.

That being said, other models, except *Mour2019CNN*, also achieve reasonably good results in all experiments. However, none of the models consistently achieves the second-best performance in both experiment settings. Even though *Saxe2015Deep* uses features from multiple scopes, they do not show better results than *Bald2013Meta* and *Mosk2008OB*. The lack of any mechanism to understand assembly code is an obvious reason, as modelling assembly code in our model improves the performance. Our improved way of representing printable string features, combined with our IFFNN, is the other reason. This is validated in the next subsection.

Mour2019CNN performs much worse than other models, even though we tried alternative hyper-parameter values in addition to the values the authors provided. One reason is that the way it represents an executable as an image is not sophisticated; even a small offset change in an executable would result in totally different textures in its image. In addition, we also observe overfitting, as its accuracy on the training set achieves 89.2%, while on the test set it is 71.8%. Even though our model is also a deep learning model, it does not suffer from the overfitting problem because we use two pre-training tasks to adequately train the Satellite-Planet Transformer and Planet-Star Transformer with the rich information embedded in assembly code. In contrast, *Mour2019CNN* can only be trained with the labels of executables, which is insufficient.

4.3.5 Interpretability

Case Study

Table 4.1 shows how our model explains the detection result of a sample. The primary factors that lead to the prediction of 05c199.exe to be malicious and the main assembly functions related to the prediction are given. It can be seen that the assembly code of the target executable is the primary reason. The two assembly functions that contribute the most to the prediction set the program to sleep for a certain time and then download and run an embedded executable from a remote address.

Qualitative Analysis

To better understand the impacts of the features we use, Table 4.7 shows the ten most frequent main factors leading to the prediction of a sample to be malware or benign.

Main factors for both classes The assembly code of an executable is one of the most frequent factors influencing the prediction of an executable to be malicious or benign. This means that the vector representing the semantic meaning of assembly code computed by our Galaxy Transformer is very effective for malware detection. We randomly examine the assembly functions of some malware that acquire the largest attention by the relay node at the top layer of the Star-Galaxy Transformer. Many of them concern malicious behaviors, such as installing itself into some registry, hijacking some common legitimate DLLs, and injecting itself into another process. We find that there are statistical differences between the two classes in the mean values of total number of strings, number of uncommon strings, total number of PE imports, and maximum entropy of the sections. To be more specific, on average there are less common strings, more uncommon strings, less PE imports, and higher entropy among malware. The fact that these features could be main factors for both classes also shows the superiority of our IFFNN over logistic regression: as the number of uncommon strings and the number of PE imports always have

Table 4.7: Most frequent main factors leading to the predictions of the malicious or benign class.

Main factors leading to the prediction of the malicious class
Assembly code
Total number of PE imports
Number of uncommon strings
The frequency of the string "Password"
The import of *KERNEL32.dll*
Total number of strings
The import of *WriteFile*
The frequency of the string "\x02\x02GetLastError"
Subsystem
Maximum entropy of sections
Main factors leading to the prediction of the benign class
Total number of strings
Number of uncommon strings
The import of *LCMapStringW*
Total number of PE imports
Assembly code
Maximum entropy of sections
The frequency of the string "\r\x01\x01\x01\x05"
The frequency of the string "\r\x01\x01\x05\x05"
The import of *initterm*
Mean entropy of sections

non-negative values, when each of them serves as a main factor leading to the prediction of the malicious class, its weight is positive (i.e., $w(v)_j v_j > 0 \& v_j > 0 \Rightarrow w(v)_j > 0$), and when it serves as a main factor leading to the prediction of the benign class, its weight is negative (i.e., $w(v)_j v_j < 0 \& v_j > 0 \Rightarrow w(v)_j < 0$). This cannot be achieved by logistic regression because when it is trained, the weight for each feature is determined and stays static, irrelevant of the input samples. However, the weight of each feature in IFFNN is dynamically computed based on the whole context, i.e., the vector representing all features.

The explanation from the perspective of statistics is as follows. All supervised machine learning classification models work by identifying the correlation between a fea-

ture and a class. Logistic regression can only learn the independent correlation between a feature and a class, without considering the correlation between features; therefore, it is linear and the weight for each feature is static. IFFNN learns the correlation between a feature and a class in a context considering the correlations between different features.

Main factors for malicious class The import of ”KERNEL32.dll” is a main factor for the prediction of malicious class because malware relies heavily on a large number of core APIs in it to manipulate memory and the file system. The ”WriteFile” function is also a main factor because malware such as ransomware and worms uses it to write content to the file system. The string of ”Password” is another main factor that more frequently appears in malware created for credential theft purposes. Malware often uses mutex for different reasons. For example, it can be used as a locking mechanism to serialize access to a resource on the system or to avoid more than one instance of itself running. ”GetLastError” is used to determine whether a mutex already exists. This is the reason why the frequency of string ”\x02\x02GetLastError” is a main factor leading to the prediction of malware.

Main factors for benign class ”LCMapStringW” is often used by benign software to convert all characters of strings to upper/lower case, which is a feature much less used in malware. ”initterm” is used by core libraries to initialize a function pointer table and does not need to be imported by software programs, and therefore it is an indicator of some benign libraries. ”*\r\x01\x01\x01\x05*” and ”*\r\x01\x01\x05\x05*” are two strings that appear 1.8 and 3.6 times respectively more frequently among benign executables than malicious executables.

Quantitative Analysis

We also use a quantitative measure to analyze the explanation of *I-MAD*. We compute the *Gini importance (GI)* and *information gain (IG)* of the features, and then rank them based

on those criteria. We then rank the features by the frequencies that they serve as the main factors for the predictions. Features serving as main factors more frequently should be relatively important features for malware detection. It should be noted that even though the importance ranked this way is relevant to the rank by Gini importance or information gain, they are not supposed to be equivalent. Even if the attribution mechanism of *I-MAD* gives a perfect explanation, the feature importance rank based on that would still be different from the rank by Gini importance or information gain.

Table 4.8 shows the Spearman's Rank Correlation Coefficient between the rank given by *I-MAD*, Gini importance, and information gain. It can be seen that the Spearman's Rank Correlation Coefficient between the rank given by *I-MAD* and those given by Gini importance and information gain are 0.59 and 0.55, respectively. This shows a strong correlation between them. The correlation coefficient between the rank by information gain and by Gini importance is only 0.72, even though they are often used for the exact same purpose: feature selection. The result means that the IFFNN in *I-MAD* frequently uses features that have high information gain or Gini importance as its main classification factors.

Table 4.8: The Spearman's Rank Correlation Coefficient between the feature importance rank given by *I-MAD*, Gini importance, and information gain.

	IG	GI	*I-MAD*
IG	1.0	0.72	0.59
GI	0.72	1.0	0.55

4.3.6 Efficiency Study

The efficiency of *I-MAD* and all models for comparison measured by the number of samples classified per second is presented in Table 4.9.

Among all models, the efficiency of *I-MAD* is moderate. And *I-MAD* is the second most efficient deep learning model. *Saxe2015Deep* is the most efficient because the dimen-

Table 4.9: Efficiency of each model in terms of number of samples classified per second. The time consumption for feature extraction is not included.

Model	n samples per second
Mosk2008OB	32,152
Bald2013Meta	127,988
Saxe2015Deep	142,711
Raff2017MalC	86
Krcal2018Conv	142
Mour2019CNN	391
SVM (same features)	58
I-MAD (no code)	28,197
I-MAD (ST)	15,355
I-MAD (ST+)	15,239

sion of its feature vector is only 1024, and the network is very small. *Raff2017MalC* and *Krcal2018Conv* are slow because they rely on whole byte sequences, and they are very computationally expensive. With our Titan Xp graphics cards their batch sizes could be around 32 and 128 at most, respectively. The batch size for *Mour2019CNN* depends on the number of bytes in the samples; in extreme cases we need to run the model on the CPU because the graphics card memory cannot hold the computation for even one large executable. For *I-MAD* (ST)/(ST+), the batch size could be at least 512 for most samples. As the representation of assembly code is computed at three levels (i.e., basic block, function, and executable), the memory for the lower level computation is released and reused after the representation is computed. For *I-MAD* (no code), the batch size could be 5,120. It is worth the extra computational cost to model assembly code because the benefit of it in classification performance is significant. SVM with the same features as *I-MAD* also has very low efficiency because its computational complexity is linear with the dimension of feature vector and the number of support vectors, which are large when the dataset is complex. In our experiments, there are always more than 43,000 support vectors, and the dimension of feature vectors is more than 2,700.

Chapter 5

Defense Against Black-box Adversarial Attacks

Defense against adversarial attacks can be broadly categorized as proactive or reactive methods [137]. The chosen method can either fortify the robustness of the machine learning model per se with embedded mechanics or special training procedures [37,48,63,110, 129,137], or it can defend the target model by recognizing adversarial samples [54,62,93, 143]. Existing defense methods including both proactive and reactive methods are static methods, which means that they cannot update their states while countering adversaries. In the black-box scenario, a successful attack requires more than thousands of queries [70]. Conversely, each query could also be used by the system to counter the attacks. Existing methods do not have the mechanics in place to utilize them, which leads to an unnecessary disadvantage of the defenders because adversaries can update their attack strategies according to the information acquired from their pry attempts. A more effective defense method that can dynamically update its state is yet to be proposed.

Online machine learning is a family of machine learning in which a model updates its state whenever it receives a new query sample, as opposed to offline learning, in which a model is trained on a fixed training set and then remains static. The advantage of online machine learning is its adaptive capability that allows it to update its state according

to the new experience. In this study, we propose *DyAdvDefender*, an instance-based online machine learning model to defend against black-box adversarial attacks. DyAdvDefender can recognize a perturbed sample that originates from the same sample as a previously queried sample, and it can output the same classification result for all the perturbed samples that have the same origin. Thus, the classification result is intact, and the adversaries cannot estimate the gradient or derivative, as the perturbations of inputs cause no change of the output. To implement this approach we propose solutions to address two main challenges: (i) how to determine whether two samples have the same origin and (ii) how to keep the time consumption of the defense mechanism in a controllable range.

We summarize the contributions of this part of research as follows:

1. We propose the **first** defense method that is based on **online machine learning** and **instance-based learning** to dynamically update its states according to received attacks.

2. We propose a novel dedicated *locality sensitive hashing* (*LSH*)-based optimization method for DyAdvDefender to index and retrieve samples with an exemplary efficiency sufficient for real-world applications.

3. We propose a new evaluation procedure to assess an online defense method because all previous evaluation procedures are only suitable for static defense methods.

4. Extensive experiments on different types of datasets suggest that our defense method, DyAdvDefender, significantly outperforms existing state-of-the-art defense methods in terms of defense effectiveness, while not compromising classification accuracy on natural samples.

5.1 Proposed Defense Method

In this section, we describe the proposed defense method: DyAdvDefender. As explained earlier, substitute model attacks and perturbation trial attacks are different methodologies

in the black-box attack scenario, and the former is a lesser threat because of its limitations and the effectiveness of existing defense mechanisms against it. We therefore focus on defending against perturbation trial-based black-box attacks.

5.1.1 Preliminaries

There are slightly different settings for black-box attacks. We follow the setting [32,70,135] in which the adversaries have no knowledge of the attacked system but have access to the complete output, including the probability that a query sample belongs to each class.

To craft an adversarial sample with perturbation trial attacks, the adversaries perturb a natural sample many times to form **prying samples** that are used to query the target model. They accumulate the effective perturbations until a successful **adversarial sample** is found. This implies that during the attack, the target model receives several prying samples that have the same **origin**, which is the natural sample. We give a formal definition of a sample's *origin* as follows.

Definition 5.1.1 (Origin)**.** Consider a natural sample $\boldsymbol{x_0}$, for any sample $\boldsymbol{x} = \boldsymbol{x_0} + \boldsymbol{\delta}$, that is crafted by perturbing $\boldsymbol{x_0}$. $\boldsymbol{x_0}$ is called the origin of $\boldsymbol{x}$. ■

Due to the nature of adversarial samples and their crafting, the perturbations of adversarial samples to their original samples should be small ($||\boldsymbol{\delta}|| \ll ||\boldsymbol{x_0}||$) for multiple reasons [134]. For example, for images, perturbations that are smaller than 1.0E-2 per pixel in L_2 measure are small enough and also sufficient to craft adversarial samples [135]. The adversaries want them to be hardly perceptible for the purposes of stealthiness and invariant semantics. From the perspective of calculus, the perturbation should be small enough so that the first-order gradient could be approximated, and thus the accumulation of the perturbations could stay effective and efficient. This property of small perturbations can be used to determine whether two samples have the same origin.

5.1.2 Overall Defense Mechanism

We present the unoptimized DyAdvDefender in Algorithm 1. When the system receives a query, it checks whether there is any indexed sample that has the same origin. If so, it outputs the prediction of the indexed sample and indexes the query with the output. If there is no indexed sample that has the same origin, it outputs the predicted result of the query by the machine learning model to defend, and it indexes the input and output pair as well.

Algorithm 1: Unoptimized DyAdvDefender.

Data: a query $\boldsymbol{x}$, a set of indexed samples S, the machine learning model to defend C
Result: Classification result y
for $(\boldsymbol{x_0}, \boldsymbol{y_0}) \in S$ **do**
  if $Same\_Origin(\boldsymbol{x_0}, \boldsymbol{x})$ *is True* **then**
    $\boldsymbol{y} = \boldsymbol{y_0}$;
    $Index((\boldsymbol{x}, \boldsymbol{y}), S)$;
    Return $\boldsymbol{y}$;
  end
end
$\boldsymbol{y} = C(\boldsymbol{x})$;
$Index((\boldsymbol{x}, \boldsymbol{y}), S)$;
Return $\boldsymbol{y}$;

We have two assertions about this system if the $Same\_Origin$ sub-algorithm performs perfectly:

1. It outputs the correct classification result if the defended model can correctly predict the origin of the query sample.

2. It does not reveal any gradient information.

We have the first assertion because the outputs for all samples that have the same origin are the result of the original sample or a sample that is randomly perturbed once; it is unlikely that one random perturbation can mislead the classifier [32, 70, 135].

We have the second assertion because the output is not affected by the perturbation on the input. Thus, the adversaries cannot estimate the gradient or the partial derivatives with respect to input variables in the black-box setting.

DyAdvDefender is an online machine learning model because its state changes as more queries are received. It is also an instance-based learning model because the prediction depends on received queries. To the best of our knowledge, this is the first defense method that is based on online machine learning or instance-based learning. Even though the idea is simple, there are two challenges to address: 1) how to determine whether two samples have the same origin, and 2) how to keep the classification procedure efficient even when a large number of samples are indexed.

5.1.3 Determination of Same Origin

Based on the property that the perturbations in prying and adversarial samples are small, we propose to recognize samples having the same origin based on their distance. Let us imagine an *ideal* distance threshold θ_0 that can be used to determine the distance between two samples as follows.

Definition 5.1.2 (Ideal Distance Threshold θ_0)**.** Let D be a distance measure. Consider two random samples $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$. Let θ_0 be a specific threshold defined on measure D such that if the distance between any two samples on measure D is smaller than θ_0, they have the same origin. If the distance is no less than θ_0, they have different origins. It can be expressed as follows:

$$Same\_Origin(\boldsymbol{x_1}, \boldsymbol{x_2}) = \begin{cases} True & D(\boldsymbol{x_1}, \boldsymbol{x_2}) < \theta_0 \\ False & D(\boldsymbol{x_1}, \boldsymbol{x_2}) \geq \theta_0 \end{cases}$$

The θ_0 is *ideal*. It is unknown whether it exists. However, we can obtain an empirical approximation of θ_0. Let us discuss the measure of distance first.

Measure of Distance

The best measure for computing the distance between two samples is determined by the nature of the field of application. For samples that can be represented as a feature vector of a fixed length, any L_p Minkowski distance could be a valid measure. If the vectors are real-valued (e.g., the greyscale values of pixels of an image), the Euclidean distance (L_2): $D(\boldsymbol{x_1}, \boldsymbol{x_2}) = ||\boldsymbol{x_1} - \boldsymbol{x_2}||_2$ is the natural choice to measure the distance of two vectors. If different features are at different scales, feature standardization should be applied on the feature vectors before computing the distance so that the value of each dimension ranges between -0.5 and 0.5. If the features represented by the vectors are integers (e.g., the frequencies of different printable strings in an executable) or binary (e.g., whether a DLL function is imported in an executable), the Manhattan distance (L_1) is the natural choice.

In the fields where samples cannot be represented as a vector but rather as a sequence, the way to measure the distance of two samples is not obvious and is another research topic. We do not elaborate on this because it is not the focus of this chapter.

Threshold of Distance

We have defined θ_0, the *ideal* distance boundary, to determine whether two samples have the same origin. Any two samples that have a distance no less than θ_0 have different origins. Conversely, any two samples that have a distance less than θ_0 have the same origin. There is no theory to indicate whether such an ideal threshold exists or how to find it. The whole concept is similar to training an artificial neural network: we suppose the input and output have an intrinsic relation described as a function of a specific neural network architecture and unknown parameters. It is unknown whether the function could resemble the underlying relation between the output and input, and it is unknown how to find the precise parameters working for all samples. We can only fit the function onto a finite set of samples (i.e., the training set) and expect it to generalize well to unseen samples that are drawn from the same distribution as the training set. This is called

empirical risk minimization (ERM) of the parameters. We also want to use a dataset to empirically approximate θ_0. Let us first identify two corollaries about θ_0.

Corollary 1. θ_0 is the minimum distance between any two samples that have different origins.

Corollary 2. θ_0 is the maximum distance between any two samples that have the same origin.

θ_0 is defined with respect to the complete set of samples in a field. The empirical approximation of it is computed based on a finite set of samples that are randomly drawn from the complete set. Corollary 2 narrates θ_0 from the perspective of samples that have the same origin. To empirically approximate θ_0 based on this corollary requires a dataset of adversarial samples. We do not have a set of adversarial samples that are drawn from the complete set of adversarial samples. Thus, this corollary is not a good theoretical support to approximate θ_0. Corollary 1 is narrated from the perspective of samples that have different origins, exactly what a training set provides us. Therefore, it can be used to empirically approximate θ_0.

In many applications, e.g., image classification, audio processing, etc., the samples in the training sets are correctly labelled natural samples. Thus, we can get the empirical approximation of θ_0 with a training set as follows. We compute the distance between every two samples in the training set. Following Corollary1, we set θ_0^*, an empirical approximation of θ_0, to be the minimal distance between any two natural samples from any classes in the training set.

In other cases, the training set contains samples that have the same origin. For instance, a training set for malware detection probably contains malware samples from the same family, and they are slightly different versions to compromise detectors. There are also benign executables that have the same origin, e.g., a Dynamically Linked Library (DLL) file that is patched for a vulnerability and its unpatched version have the same origin. Therefore, we cannot compute the minimal distance between any two samples

in such a dataset to approximate θ_0. However, as two samples that have the same origin should belong to the same class, we can get a distance threshold that is an inferior approximation of θ_0, but is as equally effective. We set θ_0^* to be the minimal distance between two samples that (1) belong to different classes in the training set and (2) are correctly classified by the machine learning model that we aim to defend. This is a less good approximation of θ_0, because it may recognize two samples that belong to the same class, and have different origins, as having the same origin. It is equally effective due to two reasons. First, it may recognize two samples in the same class that have different origins as having the same origin, and it would not recognize two samples in different classes as having the same origin, thus it would still give the correct prediction. Second, the derivative information is still not revealed to adversaries. We only compare the distances between samples that are correctly predicted by the machine learning model to defend because in these cases, the incorrectly predicted samples contain adversarial samples, abnormal samples, or mislabeled samples that could mislead the computation of θ_0^*.

5.1.4 Optimizations

We have described how DyAdvDefender handles the input and produces the output. However, the vanilla defense procedure is not efficient enough, because it unnecessarily compares a query sample with all previously queried samples. Below we describe optimization methods to improve on this.

Locality Sensitive Hashing-based Origin Search

When there are a large number of indexed samples, computing the distance between a query sample and all indexed samples is computationally expensive and less practical in real-world application. Locality sensitive hashing (LSH) has been shown to be efficient in finding sets of nearest neighbors in previous works [17,71]. Therefore, we propose a dedicated LSH-based algorithm that is different from previous works to efficiently retrieve

a subset of indexed samples that may contain the samples having the same origin as the query sample.

The LSH function family we use can be described as follows: each hash function contains a random vector $\boldsymbol{r}$ that has the same dimension as the feature vector of a sample. The value of each entry of $\boldsymbol{r}$ is independently drawn from standard Gaussian distribution. The hash value of a sample $\boldsymbol{u}$ is computed as follows:

$$h_{\boldsymbol{r}}(\boldsymbol{u}) = \begin{cases} 1 & \boldsymbol{u} \cdot \boldsymbol{r} > 0 \\ 0 & \boldsymbol{u} \cdot \boldsymbol{r} \leq 0 \end{cases}$$

It has been proven [5,59] that for vector $\boldsymbol{u}$ and vector $\boldsymbol{v}$, the probability that they have the same hash value with $h_{\boldsymbol{r}}$ is as follows:

$$Pr[h_{\boldsymbol{r}}(\boldsymbol{u}) = h_{\boldsymbol{r}}(\boldsymbol{v})] = 1 - \frac{\theta(\boldsymbol{u}, \boldsymbol{v})}{\pi} \tag{5.1}$$

Two vectors that have a small angle have a large probability to have the same hash value and vice versa. The angle between a natural sample ($\boldsymbol{x_0}$) and its perturbed version ($\boldsymbol{x} = \boldsymbol{x_0} + \boldsymbol{\delta}$, where $||\boldsymbol{\delta}|| \ll ||\boldsymbol{x_0}||$) is very small, and both tend to have the same hash values.

We use k such LSH functions to compute a signature of k bits for each sample, so there are 2^k possible different signatures. To increase the chance that those k LSH functions could evenly split different samples, we randomly generate LSH functions until we get k of them, of which the hash values of 40%$\sim$60% of training samples (i.e., natural samples) are 1, and the remaining 40%$\sim$60% are 0. We use a normal hash table to store and retrieve the set of samples corresponding to each signature. Samples that have the same origin tend to have the same signature. Given a query sample, we compute its signature and the distance between the query sample and the indexed samples that have the same signature to see if any of them has the same origin as the query sample. When there are n samples indexed, the number of indexed samples a query sample is expected to compare with is

on average $n/2^k$, in an ideal situation. This is exponentially more efficient in terms of k than comparing each query sample with all n indexed samples. For example, to compare each query with 1,024 indexed samples on average, which could be efficiently done by most GPUs for data science nowadays, only 10 LSH functions are required when there are 1 million indexed samples, and 20 LSH functions when there are 1 billion indexed samples.

As a probabilistic algorithm, there is inevitably a small probability that two samples with the same origin have different signatures. Therefore, we use m sets of k LSH functions to compute m signatures. Each indexed sample has m signatures and is put in m sets. We compute m signatures of a query sample and compare it with any sample in the union of the m sample sets corresponding to the m signatures. Empirically, we set $m = 2$ since it is sufficient to allow two samples that have the same origin to be in one set [49]. Algorithm 2 describes the procedure to find the set of samples that potentially have the same origin as a query sample.

Algorithm 2: The function to return a set of indexed samples that may have the same origin as the query sample.

Data: a query $\boldsymbol{x}$, $m \times k$ LSH functions H, m hash tables that map a signature to a set of samples HT
Result: A set of indexed samples that potentially have the same origin as the query sample S_0

```
S_0 = ∅;
i = 0;
while i < m do
    sig = new List<Bit>();
    j = 0;
    while j < k do
        h_r = H(i, j);
        sig.append(h_r(x));
    end
    S_0.union(HT_i.get(sig));
end
Return S_0;
```

Selective Indexing

Indexing every query sample gives DyAdvDefender the greatest opportunity to identify a new query sample as a perturbed version of a previous query sample. However, it is also very inefficient because an adversary may query with tens of thousands of prying samples that have the same origin [70].

Another extreme practice is to only index a query sample if DyAdvDefender cannot find an indexed sample that has the same origin. This setting yields the highest efficiency but could impair the effectiveness. When adversaries realize that perturbation does not work with our defense, they may gradually increase the scale of the perturbation, in which case DyAdvDefender may be compromised.

To keep both effectiveness and efficiency, we index two kinds of query samples. One is query samples with which no indexed samples have the same origin. The other is query samples that have the same origin as some indexed samples, but the distance between the query samples and the closest indexed samples are larger than $\theta_0^*/10$. By indexing the sample where the perturbation is one order of magnitude smaller than θ_0^*, we can invalidate the incremental attacks that may exist in practice as we mentioned above.

Optimized Defense Mechanism

Algorithm 3 presents the optimized version of DyAdvDefender that encompasses the LSH-based origin search and selective indexing.

5.1.5 Interpretability

As an instance-based defense method, DyAdvDefender has intrinsic interpretability. When it determines that an input sample can be an adversarial sample, it can explain with this template: "The received sample x can be an adversarial sample because the distance between x and our previously indexed sample x_0 is d, which is smaller than the minimal distance θ between two natural samples. Therefore, to prevent the potential revealing of

Algorithm 3: Optimized DyAdvDefender.

Data: a query $\boldsymbol{x}$, the machine learning model to defend C, a set of indexed samples S, a distance threshold θ_0^*
Result: Classification result $\boldsymbol{y}$

```
S_0 = Potential_Same_Origin(S, x);
min_sam = None;
output = None;
min_dist = ∞;
for (x_0, y_0) ∈ S_0 do
    if D(x_0, x) < min_dist then
        min_dist = D(x_0, x);
        min_sam = x_0;
        output = y_0;
    end
end
if min_dist < θ_0^* then
    y = output;
    if min_dist ≥ θ_0^*/10 then
        Index((x, y), S);
    end
    Return y;
else
    y = C(x);
    Index((x, y), S);
    Return y;
end
```

derivative information, the system will output the prediction for x_0". For validation purposes, the two samples x and x_0 can be presented to an inspector to check whether they have the same origin.

5.2 Discussion

The computational complexity of DyAdvDefender depends on the features used and the number of samples indexed. It is proportional to the sizes of the features because the feature of a query sample is compared with those of the indexed samples. The computational complexity of DyAdvDefender also depends on the feature types (e.g., integer and float) because their computational costs are different. As we stated above, the aver-

age number of indexed samples to compare with a query sample is $2^{-k}n$. If the indexed samples are distributed on z computing units, the computation of distances between the indexed samples and a query could be done by the z computing units simultaneously, with each comparing the query sample with $2^{-k}n/z$ different indexed samples. It means that z computing units could give approximately z times speedup compared with a single node. Thus, DyAdvDefender is linearly scalable.

5.3 Experiments

We conduct experiments to evaluate DyAdvDefender. Specifically, we try to answer the following research questions:

- ***RQ1***. How effective is DyAdvDefender in defending adversarial attacks compared with previous state-of-the-art defenses?
- ***RQ2***. Does DyAdvDefender affect the classification accuracy on natural samples?
- ***RQ3***. How does the average response time grow with the number of indexed samples?
- ***RQ4***. How does the number of LSH functions k affect the classification accuracy, attack success rate, and average response time?
- ***RQ5***. How does the threshold θ_0 used in the algorithm of DyAdvDefender affect the classification accuracy and attack success rate?

Our experiments are conducted on a server with one Intel(R) Core(TM) i9-9980XE CPU, 128 GB memory, and an Nvidia GeForce RTX 2080 Ti Graphics Card.

5.3.1 Datasets

We conduct experiments on two different domains of datasets: image classification and malware detection. They are representative fields with different characteristics in terms

of the constraints on perturbation. In image classification, the samples are images, and the perturbation is performed on the values of pixels. The only constraint is that the perturbed values stay in the legitimate range. In malware detection, the samples are executables. There are different choices for the feature sets, and they cannot be perturbed arbitrarily because the file format should be kept, and the logic should be intact.

The most commonly used image data sets for adversarial attack and defense research are MNIST [86] and CIFAR-10 [80], and we follow this convention. Both datasets have 10 classes of samples, and the greyscale or RBG values are standardized to the range [-0.5,0.5]. The two datasets have 10,000 samples in the test set. We reserve 500 samples from the test sets for adversarial attacks and the remaining 9,500 samples for evaluating the classification accuracy on the natural samples. The adversarial attacks are conducted on the reserved samples that are correctly classified before the attacks [54, 62, 95, 103, 135].

We collected a malware detection dataset that consists of 5,280 benign executables and 5,280 malicious executables. It has 8,448 samples in the training set, 1,056 in the validation set, and 1,056 in the test set. Following the literature [9, 45], the objective of the attack is always to trick the target system into classifying malicious executables as benign. We consider three different feature sets: PE header numerical fields, frequencies of printable strings, and imports of DLL functions [14, 88]. PE header numerical field features are standardized to the range [-0.5,0.5] because each is at a different scale. We reserve 200 malware samples from the test set for adversarial attacks. The adversarial attacks are conducted on the reserved samples that are correctly classified if not attacked.

5.3.2 Evaluation Metrics

Two main evaluation metrics are the *adversarial attack success rate* (*ASR*) and the *classification accuracy* (*ACC*) on natural samples. Following the literature, the attack success rate is the percentage of samples that are correctly classified before the perturbations by an adversarial algorithm but are misclassified after all the perturbations have been performed by an attacker. This metric answers ***RQ1*** and is our main focus because the primary ob-

jective of a defense mechanism is to reduce the attack success rate. The other metric is the classification accuracy on natural samples. This metric answers ***RQ2***. We compare the accuracy of the system defended by DyAdvDefender and the undefended system to see whether DyAdvDefender affects the classification accuracy. For other defense methods in our experiments that use different neural networks and may also be pretrained differently, the classification accuracy comparison between those systems and the system defended by DyAdvDefender is irrelevant to the scope of this study.

5.3.3 Two-Round Evaluation

In previous researches [54, 62, 95, 103], the evaluation was first performed as a "one-time attack and defense game", then the attack success rate or the classification accuracy on the adversarial samples was computed. This setting is sufficient for evaluating previous defense methods because they are static and do not change their states as more query samples arrive. To properly evaluate DyAdvDefender as an instance-based online machine learning model, we design a two-round evaluation procedure.

In the first round, we query the classification system on natural samples to evaluate its classification accuracy. Then, we clear the indexed samples of DyAdvDefender and use an adversarial algorithm to attack the system with the reserved samples for adversarial attacks to evaluate the attack success rate. The classification accuracy of the defended system is compared with the undefended one, and the difference represents whether the accuracy is affected by our defense mechanism DyAdvDefender. The attack success rate of the defended system is also compared with the undefended one, and the difference represents the effectiveness of DyAdvDefender.

The objective of the second round is to observe (i) how adversarial samples affect the defended system in terms of classification accuracy on natural samples and (ii) how natural samples affect the defense effectiveness with adversarial attacks. Therefore, in the second round, DyAdvDefender keeps the samples indexed during the adversarial attacks in the first round, and then we query the system with natural samples to evaluate

its classification accuracy. This achieves objective (i). We then clear the indexed samples and query the system with the natural samples again to let DyAdvDefender index them, and then we use an adversarial algorithm to attack the system with the reserved samples to evaluate the attack success rate. This achieves objective (ii).

It should be noted that when we evaluate the undefended system or other defense methods, we also follow a two-round evaluation. The difference is that there is no step for clearing the indexed samples for them, because they do not index anything and the states of those systems do not change in the two rounds. Hence, the classification accuracy and attack success rate in the two rounds should be approximately the same.

5.3.4 Adversarial Attack and Defense Methods

We use the following perturbation trial-based black-box adversarial attack methods to demonstrate the effectiveness of DyAdvDefender: ZOO, ZOO_AE, AutoMZOOM, and BLZOOM with their default settings to attack a deep learning model for image classification tasks [32, 135].

- ***ZOO*** is a coordinate-wise gradient estimation-based black-box attack method [32]. It perturbs one variable each time and accumulates the perturbations until a successful adversarial sample is found or a preset number of iterations has been tried.

- ***ZOO_AE*** is an improved version of ZOO [135]. It uses an autoencoder to learn a compressed representation of a sample. The perturbation is performed on the compressed representation.

- ***AutoZOOM*** is a more significantly improved version of ZOO [135]. In addition to using an autoencoder, AutoZOOM adopts an adaptive random gradient estimation strategy to improve the efficiency.

- ***BLZOOM*** is a variant of AutoZOOM [135]. It uses a bilinear resizing operation to replace the autoencoder in AutoZOOM.

We set 1.0E-2 as the per-pixel perturbation bound on images to keep their original semantics. We discuss the validity of this setting in Section 5.3.11.

The integrity of the PE format and the original functionalities need to be kept when perturbing malware samples, and therefore there are constraints on the perturbations [9]. A perturbation on a compressed representation corresponds to the changes of multiple features, so the constraints cannot be guaranteed. Therefore, we can only use ZOO to attack a deep learning classifier for malware detection while following the constraints. As the constraints have already guaranteed that the original functionalities are kept, we do not set a perturbation bound for malware samples.

We compare DyAdvDefender with the following state-of-the-art defense methods [4, 62, 95, 103]:

- ***Grosse et al.*** [62] augment a classification model with an additional class for adversarial samples and train the model to classify adversarial examples to that class. We implement their approach and apply it in the experiments with all datasets.

- ***Akhtar et al.*** [4] learns a Perturbation Rectifying Network (PRN) to reconstruct an image from a potentially perturbed one. Then, a perturbation detector is trained on the Discrete Cosine Transform of the input-output difference of the PRN. If the detector determines that an image is perturbed, the reconstructed image is fed to the original classifier; otherwise, the original image is fed.

- ***Madry et al.*** [95] prove that projected gradient descent (PGD) is the strongest first order white-box attack and propose an objective function to train a network to be robust against PGD attacks so that it can also defend against other inferior attacks. The authors provide the pre-trained robust models for the MNIST and CIFAR-10 datasets, which are different from the models defended by DyAdvDefender, so we include the comparison on the reduction of the attack success rate with their methods on those two datasets.

- ***Musta et al.*** [103] propose a new objective function called Prototype Conformity Loss (PCL) to train a neural network. It forces the features of samples in different classes to have a large distance with each other, which makes the adversarial perturbations more difficult to fulfill their objectives. The authors provide the source code only for the experiments with CIFAR-10 and use a different neural network; therefore, we compare the reduction of the attack success rate with their defense only on CIFAR-10.

5.3.5 Classification Models to Defend

Table 5.1: The architectures of the neural networks defended by DyAdvDefender for malware detection.

Feature	Header	Strings	Imports
Input	54	4,651	10,201
Layer 1	$Relu(FC(32))$	$Relu(FC(64))$	$Relu(FC(128))$
Layer 2	$Relu(FC(16))$	$Relu(FC(32))$	$Relu(FC(64))$
Layer 3	$Softmax(FC(2))$	$Softmax(FC(2))$	$Softmax(FC(2))$

For the undefended classification system, the classification system defended by DyAdvDefender, and the system defended by the method proposed by Grosse et al. [62], the target models on the two image datasets are the two convolutional neural networks used in [135]. We directly download their pretrained models for our experiments.

For the other defense methods that we compare with [95, 103], the networks are given in their papers and source code. We use the models that they provide to conduct the experiments. Because they use different neural networks, the comparison with those models on classification accuracy on natural samples is not relevant to the performance of defense mechanisms, and we make our comparison with them only on the reduction of the attack success rate.

The undefended models for malware detection are feed-forward neural networks with two hidden layers applied on the three feature sets respectively. They are trained with cross entropy as the objective function and Adam as the optimization algorithm. The details of the architectures are presented in Table 5.1.

5.3.6 Hyperparameters

The distance threshold θ_0^* values computed on the training sets are shown in Table 5.2.

Table 5.2: The distance threshold θ_0^* computed on each dataset and each feature set.

Dataset (Feature)	Distance measure	θ_0^*
MNIST	Euclidean	0.82
CIFAR-10	Euclidean	2.75
Malware Detection (header)	Euclidean (Standardized)	0.28
Malware Detection (imports)	Manhattan	2
Malware Detection (strings)	Manhattan	8

We set the number of LSH functions $k = 10$ in our basic experiments, which is sufficient for even larger datasets than the ones we used in the experiments. We also show how the changes of k affect the performance in all respects in Section 5.3.8.

5.3.7 Results

As the adversarial algorithms have randomness, we run each experiment three times and report the mean. The classification accuracy on natural samples and attack success rate in the two rounds are presented in Table 5.3 for MNIST, Table 5.4 for CIFAR-10, and Table 5.5 for malware detection.

As shown, all defense methods could reduce the ASR of all black-box attack methods to different degrees in the experiments. Overall, our defense method DyAdvDefender significantly outperforms all other defense mechanisms in reducing the ASR, especially for MNIST, where the ASR is reduced to almost 0%. Akhtar et al. [4] is the runner-up

Table 5.3: Experiment results on MNIST.

Attack	Defense	Round 1		Round 2	
		ACC	ASR	ACC	ASR
ZOO	No defense	99.4%	99.0%	99.4%	99.0%
	DyAdvDefender	99.4%	**0%**	99.4%	**0%**
	Akhtar et al.	99.4%	9.1%	99.4%	8.9 %
	Grosse et al.	99.2%	77.6%	99.2%	77.3%
	Madry et al.	98.4%	9.2%	98.4%	9.5%
ZOO_AE	No defense	99.4%	99.2%	99.4%	99.2%
	DyAdvDefender	99.4%	**0%**	99.4%	**0%**
	Akhtar et al.	99.4%	**0%**	99.4%	**0%**
	Grosse et al.	99.2%	81.3%	99.2%	81.7%
	Madry et al.	98.4%	99.3%	98.4%	99.4%
BLZOOM	No defense	99.4%	99.2%	99.4%	99.4%
	DyAdvDefender	99.4%	**0%**	99.4%	**0.1%**
	Akhtar et al.	99.4%	1.6%	99.4%	1.7%
	Grosse et al.	99.2%	51.6%	99.2%	51.3%
	Madry et al.	98.4%	64.4%	98.4%	64.6%
AUTOZOOM	No defense	99.4%	99.6%	99.4%	99.4%
	DyAdvDefender	99.4%	**0.2%**	99.4%	**0%**
	Akhtar et al.	99.4%	2.2%	99.4%	2.2%
	Grosse et al.	99.2%	68.1%	99.2%	68.3%
	Madry et al.	98.4%	80.4%	98.4%	80.2%

defense method. It has the same effectiveness as DyAdvDefender in reducing the ASR to defend against ZOO_AE and BLZOOM on MNIST, and it is only inferior in some other settings. However, it is much less effective in defending against three kinds of attacks (i.e., ZOO, ZOO_AE, AUTOZOOM) on CIFAR-10. The other defense methods have a larger gap with DyAdvDefender in the defense effectiveness. The unique characteristic of DyAdvDefender as an online machine learning and instanced-based model allows it to dynamically adapt itself to the coming attacks, and thus it outperforms static methods. This result provides an answer to ***RQ1***.

Table 5.4: Experiment results on CIFAR-10.

Attack	Defense	Round 1		Round 2	
		ACC	ASR	ACC	ASR
ZOO	No defense	77.9%	98.5%	77.9%	98.5%
	DyAdvDefender	77.9%	**0%**	77.9%	**0%**
	Akhtar et al.	77.9%	81.1%	77.9%	81.0%
	Grosse et al.	72.8%	98.2%	72.8%	98.2%
	Madry et al.	87.3%	71.4%	87.3%	71.5%
	Musta et al.	89.0%	75.5%	89.0%	75.5%
ZOO_AE	No defense	77.9%	98.8%	77.9%	98.8%
	DyAdvDefender	77.9%	**1.3%**	77.9%	**1.2%**
	Akhtar et al.	77.9%	79.6%	77.9%	79.9%
	Grosse et al.	72.8%	97.9%	72.8%	98.0%
	Madry et al.	87.3%	98.6%	87.3%	98.4%
	Musta et al.	89.0%	52.3%	89.0%	52.5%
BLZOOM	No defense	77.9%	99.5%	77.9%	99.3%
	DyAdvDefender	77.9%	**2.7%**	77.9%	**2.9%**
	Akhtar et al.	77.9%	14.6%	77.9%	14.6 %
	Grosse et al.	72.8%	95.2%	72.8%	94.8%
	Madry et al.	87.3%	86.5%	87.3%	86.3%
	Musta et al.	89.0%	45.1%	89.0%	45.5%
AUTOZOOM	No defense	77.9%	99.8%	77.9%	99.8%
	DyAdvDefender	77.9%	**14.2%**	77.9%	**14.2%**
	Akhtar et al.	77.9%	39.7%	77.9%	39.5%
	Grosse et al.	72.8%	97.1%	72.8%	97.2%
	Madry et al.	87.3%	94.6%	87.3%	94.6%
	Musta et al.	89.0%	37.5%	89.0%	36.8%

On natural samples, the ACC of a classification system defended by DyAdvDefender is the same as the undefended one. This suggests that DyAdvDefender does not affect the classification accuracy on natural samples and provides an answer to ***RQ2***. The defense method proposed by Akhtar et al. [4] is applied to the same classification model, and we can see that their method does not affect the ACC either. It should be noted that the ACC for the other defense methods do not change in the two rounds because the classification model to protect is a deterministic algorithm, and the ASR may change because the adversarial algorithms are randomized.

Table 5.5: Experimental results on malware detection. The attack method is ZOO.

Feature	Defense	Round 1		Round 2	
		ACC	ASR	AC	ASR
Header	No defense	96.8%	100%	96.8%	100%
	DyAdvDefender	96.8%	**73.5%**	96.8%	**73.5%**
	Grosse et al.	96.2%	99.0%	96.2%	99.0%
Imports	No defense	98.8%	100%	98.8%	100%
	DyAdvDefender	98.8%	**40.0%**	98.8%	**39.3%**
	Grosse et al.	98.5%	99.8%	98.5%	99.7%
Strings	No defense	98.5%	100%	98.5%	100%
	DyAdvDefender	98.5%	**6.0%**	98.5%	**5.3%**
	Grosse et al.	98.2%	71.2%	98.2%	70.8%

One observation is that the effectiveness of the defense methods depends on the dataset and adversarial algorithm. CIFAR-10 is harder to defend than MNIST for all defenses. Yet, there is no such consistency in adversarial algorithms. Compared with ZOO and ZOO_AE, AutoZOOM and BLZOOM are harder for DyAdvDefender to defend against but easier for the defenses proposed by Grosse et al. [62] and Mustafa et al. [103]. Madry et al. [95] defend against ZOO best and ZOO_AE the worst. Akhtar et al. [4] defend against all attacks well on MNIST and defend against AutoZOOM and BLZOOM well on CIFAR-10.

On malware detection, the defense effectiveness of DyAdvDefender and the defense proposed by Grosse et al. [62] varies from one feature set to another, e.g., they both defend better with printable strings as the features. This confirms our previous statement that the choice of the feature set could make a significant difference.

5.3.8 Efficiency Study

The overhead that DyAdvDefender brings to the classification system mainly comes from the procedure to find the indexed samples that potentially have the same origin as a query sample. Therefore, theoretically the average response time (ART) grows with the num-

ber of samples indexed in the system. That is why we use the LSH-based algorithm to optimize the efficiency. We used all the samples of MNIST and CIFAR-10 to evaluate DyAdvDefender with respect to its efficiency. Figure 5.1 depicts the relation between the average response time and the number of indexed samples. As expected, from Figure 5.1 1.a) and 2.a), we can see that the average response time of DyAdvDefender without LSH grows linearly with the number of indexed samples (i.e., $ART_{NoLSH}(N) = k_1 N + b_1$). This is because we need to compute the distance between all indexed samples and a query sample. In comparison, the growth of the average response time of the optimized DyAdvDefender with LSH can be ignored. To see it more clearly, we show the growth of the logarithm of the average response time for DyAdvDefender with and without LSH, together with the average response time of the classification system without defense in Figure 5.1 1.b) and 2.b). We can see that the logarithm of the average response time of DyAdvDefender with LSH stays at the same magnitude as the number of indexed samples grows. This means that the overhead brought by DyAdvDefender is well bounded as the number of indexed samples grows. Even though the overhead is still greater than the parametric defense methods as opposed to instance-based methods, the advantage of our defense effectiveness is worth the bounded overload. Thus, it is ideally practical for real-world applications, and as a result we have answered ***RQ3***.

5.3.9 Impacts of Number of LSH Functions

In Figure 5.2, we present the impacts of the number of LSH functions k to the average response time, classification accuracy, and attack success rate of the classification system defended by DyAdvDefender. We use ZOO and AutoZOOM as the adversarial algorithms on CIFAR-10 in these experiments, because they are the best and worst defended adversarial algorithms by DyAdvDefender. We observe that within the range we test, from 1 LSH function to 20 LSH functions, the average response time drops significantly before it reaches 10 LSH functions. In contrast, the classification accuracy on natural samples is not affected at all. As the number of LSH functions increases, the attack success

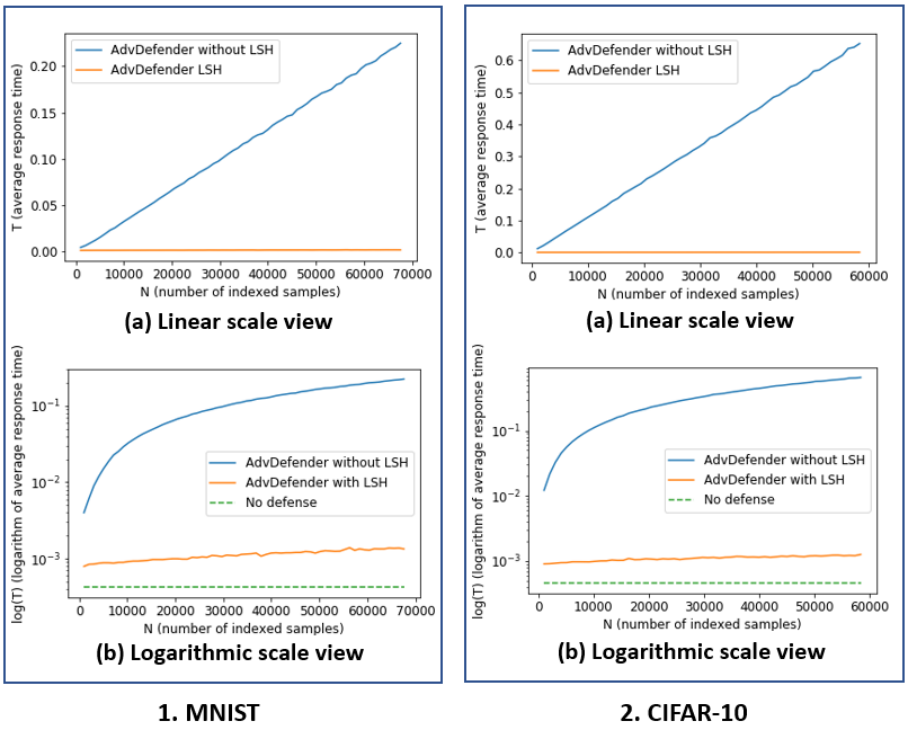

Figure 5.1: The relation between the average response time and the number of indexed samples on MNIST and CIFAR-10. The average response time is shown on both linear scale view and logarithmic scale view.

rate is not affected with ZOO and grows very slowly with AutoZOOM. The defense effectiveness is still excellent even when the number of LSH functions is 20, which keeps the average number of indexed samples to compare with a query sample at 1,024 when 1 billion samples are indexed. Therefore, depending on the number of indexed samples in real-world scenarios, classification service providers can improve efficiency by increasing the number of LSH functions without worrying about a significant drop of defense effectiveness within a large range. This provides an answer to ***RQ4***.

5.3.10 Impacts of Threshold θ_0

Figure 5.3 depicts the impacts of the distance threshold θ_0 that is set in DyAdvDefender to ASR and ACC. We can see that there is a range of θ_0 that keeps DyAdvDefender at its optimal performance (i.e., without loss of ACC and at its lowest ASR). Such an optimal

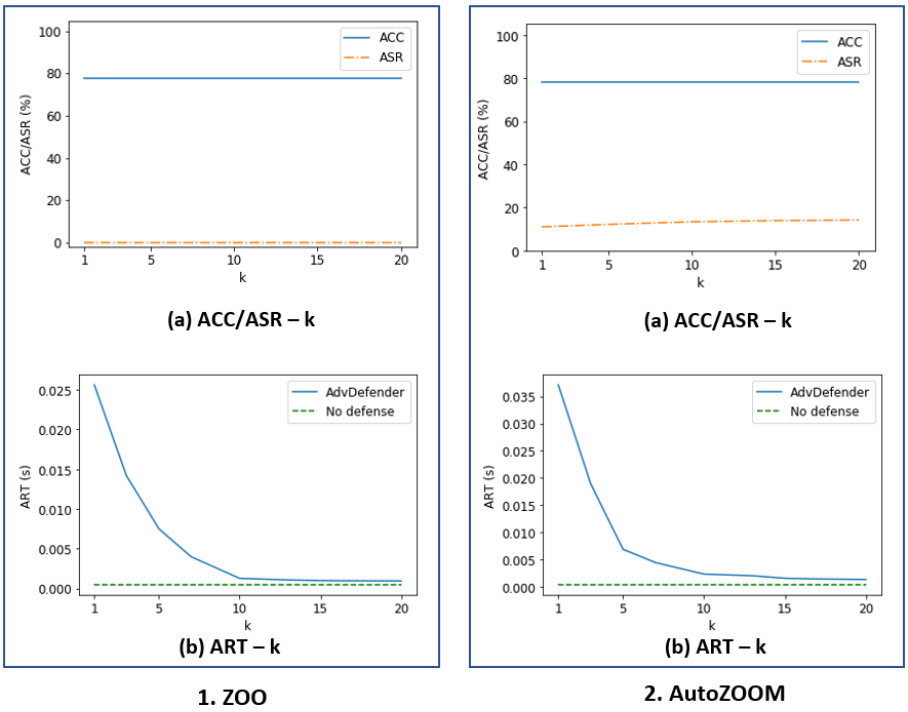

Figure 5.2: The relation between the classification accuracy, attack success rate, and average response time with the number of LSH functions, with ZOO and AutoZOOM as the attacks.

range exists because samples that have small distances are from the same classes, and even if the defense mechanism determines them as from the same origin, the classification result would not be affected. As can be seen, the empirical θ_0^* computed on the training sets is always within the optimal range.

As θ_0 decreases beyond the optimal range, the ASR increases, which means DyAdvDefender fails to defend against the attacks more frequently. This is because when θ_0 is set to be smaller than the perturbation, DyAdvDefender would consider that the perturbed sample and the original sample have different origins, thus the attack bypasses the defense mechanism. Yet, this does not affect the classification on original samples. As θ_0 increases beyond the optimal range, we can see the ASR increases and the ACC decreases. This is because as the θ_0 gets larger, samples in different classes are considered having the same origin by DyAdvDefender, so it outputs wrong classification results which negatively impacts both ASR and ACC. This result answers ***RQ5***.

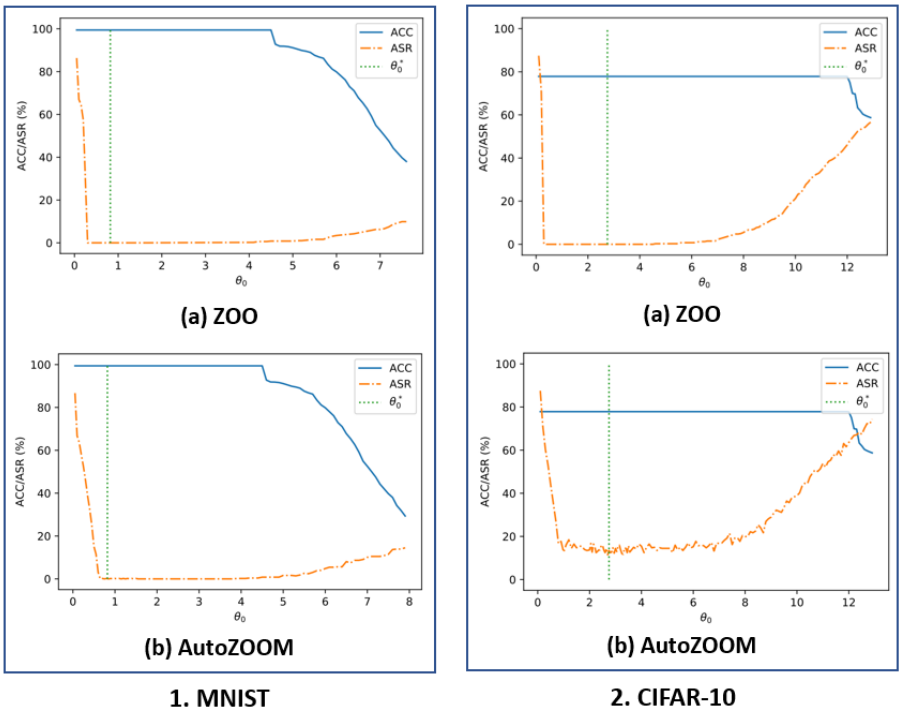

Figure 5.3: The relation between ASR/ACC and θ_0 on MNIST and CIFAR-10. The empirical distance threshold θ_0^* computed on the training sets is shown as vertical lines.

5.3.11 Validity of Adversarial Samples

We ensure that the adversarial malware samples keep their original semantics by following the rules of perturbation. However, the adversarial samples of images cannot be verified with rules. Therefore, we manually examine the adversarial samples crafted by attacking the defended methods. We can still correctly recognize all their original classes, even though for some of them the perturbations are clearly perceptible. Figure 5.4 shows some examples of successful attacks with the per-pixel perturbation between 1.0E-3 and 1.0E-2. We can see that the adversarial samples of MNIST contain some noise points, and those of CIFAR-10 seem to contain a "melted" area, but they still keep their original semantics. However, further perturbations may begin to change the semantics of the adversarial samples. This result suggests that our setting of the perturbation threshold is valid.

Figure 5.4: Adversarial samples with per-pixel perturbation between 1.0E-3 and 1.0E-2 and their original samples.

5.4 Limitations

The value of θ_0^* is computed based on a training set. As demonstrated in our experiments, θ_0^* that is computed with our approach is within the range that corresponds to the best defense performance. This is because the datasets for research use are usually large enough and clean. They are the requirements for the proposed defense method to work at its best performance. However, in real-world applications, a training set may not be perfectly curated, e.g., adversarial samples and wrong labelled samples may exist, and the training set may not be large enough. Then, the θ_0^* may go out of the most effective range. Therefore, in real-world applications a manual examination of the samples with the minimal distances and adjustment on θ_0^* may be needed so that DyAdvDefender can achieve its optimal performance.

The proposed defense mechanism DyAdvDefender relies on the comparison of feature vectors of a query sample and the indexed samples. A bad choice of the feature set may cause the defense mechanism to fail. As an extreme example, with a bad choice of the feature set, two samples from different classes could have the exact feature vectors. The machine learning classification model would not even be able to differentiate them, and the DyAdvDefender would fail too. If the machine learning model to defend is based on multiple feature sets, it is better to deploy DyAdvDefender on each feature set separately, rather than uniting all feature sets as one set, because the former could remove the effects of the difference on the distance scale across different feature sets.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we present novel solutions for interpretable classification, malware detection, and black-box adversarial defense.

For interpretable classification, we propose an intrinsically interpretable feedforward neural network (*IFFNN*) architecture. It takes tensors of a fixed shape as its input and can explain the contribution of each feature to a classification result. We also conduct comprehensive experiments to evaluate the classification performance and interpretability of the IFFNNs. We reached the conclusion that the IFFNNs achieve similar classification accuracy as their non-interpretable feedforward neural network counterparts and provide meaningful explanations with very little computational overhead. Therefore, the generalized IFFNN architecture is an excellent choice for real-world applications when explanations for classification results are expected for various reasons. The IFFNN also serves as an example to show that a machine learning model can be interpretable without sacrificing classification performance.

For malware detection, we present *I-MAD*, a novel neural network architecture that is based on the analysis on multiple feature scopes, including assembly code, printable strings, PE imports, and PE header numerical features. An advantage of I-MAD as com-

pared with previous methods is that it can model complete assembly code of a sample with its *Galaxy Transformer* component. In addition to its excellent detection performance, it can also provide explanations for its detection results, which can help malware analysts examine the results and find consistent patterns in malware samples.

To counter black-box adversarial attacks, we propose a state-of-the-art online machine learning and instance-based defense method, namely *DyAdvDefender*. As an online machine learning and instance-based defense method, it can update its states to adapt to attacks it has received. Experimental results suggest that DyAdvDefender outperforms previous state-of-the-art defense methods against perturbation trial-based black-box adversarial attacks without harming the classification accuracy on natural samples. To optimize its efficiency, we propose an LSH-based solution for indexing and retrieving samples. Experiment results also illustrate that DyAdvDefender is efficient for practical use.

6.2 Future Work

For multi-class classification with the IFFNN architecture we propose, the penultimate layer is required to have the output dimension of $c \times m$, where c is the number of classes, and m is the number of input features. It can be very large when there are many classes. If the output of the previous layer has a much smaller dimension, the performance of the neural network can be bad. As a new architecture of feedforward neural network, it is yet to be explored how to optimize the hyper-parameters of IFFNNs to make them approach their optimal performance. In addition, the proposed IFFNN can be easily generalized for regression tasks. A comprehensive evaluation on the interpretability for regression is yet to be performed.

For our malware detection method I-MAD, other pretraining methods could be proposed to train the components of the Galaxy Transformer. It can also be explored to see if the vector representation of assembly code learned with the Galaxy Transformer can be

applied for other tasks, such as vulnerability detection and function description generation.

In our proposed black-box adversarial defense method DyAdvDefender, the threshold to determine whether two samples have the same origin is computed globally for samples of all classes. In real-world scenarios, the minimal distance between two samples from different classes could be different. Further study on a variant of the proposed DyAdvDefender with different thresholds for different classes can be a direction to explore. In addition to the optimizations we have described, there could be others to be proposed when the system is deployed in real-world application scenarios. For example, the indexing database of DyAdvDefender could be cleared once a day or whenever the number of indexed samples reaches a certain value. That would barely impair the effectiveness of DyAdvDefender because the adversary could only get information at the first query after each database clearing, and a successful attack requires at least hundreds or thousands of informative queries.

Bibliography

[1] Bogus control flow. https://github.com/obfuscator-llvm/obfuscator/wiki/Bogus-Control-Flow. Accessed: 2018-08-17.

[2] Malware numbers 2017. https://www.gdatasoftware.com/blog/2018/03/30610-malware-number-2017. Accessed: 2018-08-17.

[3] AGRAWAL, R., SRIKANT, R., ET AL. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB* (1994), vol. 1215, pp. 487–499.

[4] AKHTAR, N., LIU, J., AND MIAN, A. Defense against universal adversarial perturbations. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2018), pp. 3389–3398.

[5] ALEXANDR, A., AND PIOTR, I. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Communications of The ACM* (2008).

[6] ALTMAN, N. S. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician 46*, 3 (1992), 175–185.

[7] ANDERSON, B., QUIST, D., NEIL, J., STORLIE, C., AND LANE, T. Graph-based malware detection using dynamic analysis. *Journal in computer Virology 7*, 4 (2011), 247–258.

[8] ANDERSON, B., STORLIE, C., AND LANE, T. Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence* (2012), ACM, pp. 3–14.

[9] ANDERSON, H. S., KHARKAR, A., FILAR, B., AND ROTH, P. Evading machine learning malware detection. *black Hat* (2017).

[10] ARRIETA, A. B., DÍAZ-RODRÍGUEZ, N., DEL SER, J., BENNETOT, A., TABIK, S., BARBADO, A., GARCÍA, S., GIL-LÓPEZ, S., MOLINA, D., BENJAMINS, R., ET AL. Explainable artificial intelligence (xai): Concepts, taxonomies, opportunities and challenges toward responsible ai. *Information Fusion 58* (2020), 82–115.

[11] ATHALYE, A., CARLINI, N., AND WAGNER, D. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420* (2018).

[12] BAEZA-YATES, R., AND NAVARRO, G. Fast approximate string matching in a dictionary. In *spire* (1998), IEEE, p. 0014.

[13] BAI, T., ZHAO, J., ZHU, J., HAN, S., CHEN, J., AND LI, B. Ai-gan: Attack-inspired generation of adversarial examples. *arXiv preprint arXiv:2002.02196* (2020).

[14] BALDANGOMBO, U., JAMBALJAV, N., AND HORNG, S.-J. A static malware detection system using data mining methods. *arXiv preprint arXiv:1308.2831* (2013).

[15] BALUJA, S., AND FISCHER, I. Adversarial transformation networks: Learning to generate adversarial examples. *arXiv preprint arXiv:1703.09387* (2017).

[16] BARAKAT, N., AND BRADLEY, A. P. Rule extraction from support vector machines: a review. *Neurocomputing 74*, 1-3 (2010), 178–190.

[17] BAWA, M., CONDIE, T., AND GANESAN, P. Lsh forest: self-tuning indexes for similarity search. In *Proceedings of the 14th International Conference on World Wide Web* (2005), ACM, pp. 651–660.

[18] BAYER, U., MOSER, A., KRUEGEL, C., AND KIRDA, E. Dynamic analysis of malicious code. *Journal in Computer Virology 2*, 1 (2006), 67–77.

[19] BOSER, B. E., GUYON, I. M., AND VAPNIK, V. N. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory* (1992), ACM, pp. 144–152.

[20] BRÉMAUD, P. *Markov chains: Gibbs fields, Monte Carlo simulation, and queues*, vol. 31. Springer Science & Business Media, 2013.

[21] BROWN, T. B., MANN, B., RYDER, N., SUBBIAH, M., KAPLAN, J., DHARIWAL, P., NEELAKANTAN, A., SHYAM, P., SASTRY, G., ASKELL, A., ET AL. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).

[22] BYERLY, A., KALGANOVA, T., AND DEAR, I. A branching and merging convolutional network with homogeneous filter capsules. *arXiv preprint arXiv:2001.09136* (2020).

[23] CARLINI, N., AND WAGNER, D. Towards evaluating the robustness of neural networks. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)* (2017), IEEE, pp. 39–57.

[24] CARUANA, R., LAWRENCE, S., AND GILES, C. L. Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In *Advances in Neural Information Processing Systems* (2001), pp. 402–408.

[25] CARUANA, R., LOU, Y., GEHRKE, J., KOCH, P., STURM, M., AND ELHADAD, N. Intelligible models for healthcare: Predicting pneumonia risk and hospital 30-day readmission. In *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining* (2015), pp. 1721–1730.

[26] CERNA, A. E. U., PATTICHIS, M., VANMAANEN, D. P., JING, L., PATEL, A. A., STOUGH, J. V., HAGGERTY, C. M., AND FORNWALT, B. K. Interpretable neural networks for predicting mortality risk using multi-modal electronic health records. *arXiv preprint arXiv:1901.08125* (2019).

[27] CESARE, S., AND XIANG, Y. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107* (2010), Australian Computer Society, Inc., pp. 61–70.

[28] CHAKRABORTY, A., ALAM, M., DEY, V., CHATTOPADHYAY, A., AND MUKHOPADHYAY, D. Adversarial attacks and defences: A survey. *arXiv preprint arXiv:1810.00069* (2018).

[29] CHARLAND, P., FUNG, B. C. M., AND FARHADI, M. R. Clone search for malicious code correlation. In *Proc. of the NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111)* (Koblenz, Germany, September 2012), pp. 1.1–1.12.

[30] CHEN, J., ALALFI, M. H., DEAN, T. R., AND ZOU, Y. Detecting android malware using clone detection. *Journal of Computer Science and Technology 30*, 5 (2015), 942–956.

[31] CHEN, K., ZHU, H., YAN, L., AND WANG, J. A survey on adversarial examples in deep learning. *Journal on Big Data 2*, 2 (2020), 71.

[32] CHEN, P.-Y., ZHANG, H., SHARMA, Y., YI, J., AND HSIEH, C.-J. Zoo: Zeroth order optimization based black-box attacks to deep neural networks without training substitute models. In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security* (2017), ACM, pp. 15–26.

[33] CHEN, Y., AND WANG, J. Z. Support vector learning for fuzzy rule-based classification systems. *IEEE Transactions on fuzzy systems 11*, 6 (2003), 716–728.

[34] CHOI, E., BAHADORI, M. T., KULAS, J. A., SCHUETZ, A., STEWART, W. F., AND SUN, J. Retain: An interpretable predictive model for healthcare using reverse time attention mechanism. *arXiv preprint arXiv:1608.05745* (2016).

[35] CHRISTODORESCU, M., AND JHA, S. Static analysis of executables to detect malicious patterns. Tech. rep., WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, 2006.

[36] CHUNG, F. R., AND GRAHAM, F. C. *Spectral graph theory*. No. 92. American Mathematical Soc., 1997.

[37] CISSE, M., BOJANOWSKI, P., GRAVE, E., DAUPHIN, Y., AND USUNIER, N. Parseval networks: Improving robustness to adversarial examples. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70* (2017), JMLR. org, pp. 854–863.

[38] COHEN, W. W. Learning trees and rules with set-valued features. In *AAAI/IAAI, Vol. 1* (1996), pp. 709–716.

[39] CORDY, J. R., AND ROY, C. K. The nicad clone detector. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on* (2011), IEEE, pp. 219–220.

[40] CYBENKO, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems 2*, 4 (1989), 303–314.

[41] DAHL, G. E., STOKES, J. W., DENG, L., AND YU, D. Large-scale malware classification using random projections and neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on* (2013), IEEE, pp. 3422–3426.

[42] DAI, J., GUHA, R. K., AND LEE, J. Efficient virus detection using dynamic instruction sequences. *JCP 4*, 5 (2009), 405–414.

[43] DAS, A., AND RAD, P. Opportunities and challenges in explainable artificial intelligence (xai): A survey. *arXiv preprint arXiv:2006.11371* (2020).

[44] DATTA, A., SEN, S., AND ZICK, Y. Algorithmic transparency via quantitative input influence: Theory and experiments with learning systems. In *2016 IEEE symposium on security and privacy (SP)* (2016), IEEE, pp. 598–617.

[45] DEMETRIO, L., BIGGIO, B., LAGORIO, G., ROLI, F., AND ARMANDO, A. Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583* (2019).

[46] DEMONTIS, A., MELIS, M., BIGGIO, B., MAIORCA, D., ARP, D., RIECK, K., CORONA, I., GIACINTO, G., AND ROLI, F. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing* (2017).

[47] DEVLIN, J., CHANG, M.-W., LEE, K., AND TOUTANOVA, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[48] DHILLON, G. S., AZIZZADENESHELI, K., LIPTON, Z. C., BERNSTEIN, J., KOSSAIFI, J., KHANNA, A., AND ANANDKUMAR, A. Stochastic activation pruning for robust adversarial defense. *arXiv preprint arXiv:1803.01442* (2018).

[49] DING, S. H. H., FUNG, B. C. M., AND CHARLAND, P. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22nd ACM International Conference on Knowledge Discovery and Data Mining (SIGKDD)* (2016), pp. 461–470.

[50] DING, S. H. H., FUNG, B. C. M., AND CHARLAND, P. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)* (2019), IEEE, pp. 472–489.

[51] DONG, L., YANG, N., WANG, W., WEI, F., LIU, X., WANG, Y., GAO, J., ZHOU, M., AND HON, H.-W. Unified language model pre-training for natural language

understanding and generation. In *Advances in Neural Information Processing Systems* (2019), pp. 13063–13075.

[52] FARHADI, M. R., FUNG, B. C., FUNG, Y. B., CHARLAND, P., PREDA, S., AND DEBBABI, M. Scalable code clone search for malware analysis. *Digital Investigation 15* (2015), 46–60.

[53] FARHADI, M. R., FUNG, B. C. M., CHARLAND, P., AND DEBBABI, M. BinClone: Detecting code clones in malware. In *Proc. of the 8th IEEE International Conference on Software Security and Reliability (SERE)* (San Francisco, CA, June 2014), IEEE Reliability Society, pp. 78–87.

[54] FEINMAN, R., CURTIN, R. R., SHINTRE, S., AND GARDNER, A. B. Detecting adversarial samples from artifacts. *arXiv preprint arXiv:1703.00410* (2017).

[55] FORCE, U. A. Analysis of the intel pentium's ability to support a secure virtual machine monitor. In *Proceedings of the... USENIX Security Symposium. USENIX Association* (2000), p. 129.

[56] FORET, P., KLEINER, A., MOBAHI, H., AND NEYSHABUR, B. Sharpness-aware minimization for efficiently improving generalization. *arXiv preprint arXiv:2010.01412* (2020).

[57] FREDRIKSON, M., JHA, S., CHRISTODORESCU, M., SAILER, R., AND YAN, X. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Security and Privacy (SP), 2010 IEEE Symposium on* (2010), IEEE, pp. 45–60.

[58] FREUND, Y., SCHAPIRE, R. E., ET AL. Experiments with a new boosting algorithm. In *Icml* (1996), vol. 96, Citeseer, pp. 148–156.

[59] GOEMANS, M. X., AND WILLIAMSON, D. P. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *Journal of the ACM (JACM) 42*, 6 (1995), 1115–1145.

[60] GOLDSTEIN, A., KAPELNER, A., BLEICH, J., AND PITKIN, E. Peeking inside the black box: Visualizing statistical learning with plots of individual conditional expectation. *journal of Computational and Graphical Statistics 24*, 1 (2015), 44–65.

[61] GOODFELLOW, I. J., SHLENS, J., AND SZEGEDY, C. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[62] GROSSE, K., MANOHARAN, P., PAPERNOT, N., BACKES, M., AND MCDANIEL, P. On the (statistical) detection of adversarial examples. *arXiv preprint arXiv:1702.06280* (2017).

[63] GU, S., AND RIGAZIO, L. Towards deep neural network architectures robust to adversarial examples. *arXiv preprint arXiv:1412.5068* (2014).

[64] GUO, C., RANA, M., CISSE, M., AND VAN DER MAATEN, L. Countering adversarial images using input transformations. *arXiv preprint arXiv:1711.00117* (2017).

[65] GUO, Q., QIU, X., LIU, P., SHAO, Y., XUE, X., AND ZHANG, Z. Star-transformer. *arXiv preprint arXiv:1902.09113* (2019).

[66] HE, K., ZHANG, X., REN, S., AND SUN, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition* (2016), pp. 770–778.

[67] HINTON, G., VINYALS, O., AND DEAN, J. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531* (2015).

[68] HOCHREITER, S., AND SCHMIDHUBER, J. Long short-term memory. *Neural computation 9*, 8 (1997), 1735–1780.

[69] HUANG, W., AND STOKES, J. W. Mtnet: a multi-task neural network for dynamic malware classification. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (2016), Springer, pp. 399–418.

[70] ILYAS, A., ENGSTROM, L., ATHALYE, A., AND LIN, J. Black-box adversarial attacks with limited queries and information. *arXiv preprint arXiv:1804.08598* (2018).

[71] INDYK, P., AND MOTWANI, R. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of Computing* (1998), ACM, pp. 604–613.

[72] ISLAM, R., TIAN, R., BATTEN, L. M., AND VERSTEEG, S. Classification of malware based on integrated static and dynamic features. *Journal of Network and Computer Applications 36*, 2 (2013), 646–656.

[73] JAEGLE, A., GIMENO, F., BROCK, A., ZISSERMAN, A., VINYALS, O., AND CARREIRA, J. Perceiver: General perception with iterative attention. *arXiv preprint arXiv:2103.03206* (2021).

[74] JENSEN, F. V. *An introduction to Bayesian networks*, vol. 210. UCL press London, 1996.

[75] JIANG, H., HE, P., CHEN, W., LIU, X., GAO, J., AND ZHAO, T. Smart: Robust and efficient fine-tuning for pre-trained natural language models through principled regularized optimization. *arXiv preprint arXiv:1911.03437* (2019).

[76] KINGMA, D. P., AND BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[77] KOLOSNJAJI, B., ZARRAS, A., WEBSTER, G., AND ECKERT, C. Deep learning for classification of malware system call sequences. In *Australasian Joint Conference on Artificial Intelligence* (2016), Springer, pp. 137–149.

[78] KOLTER, J. Z., AND MALOOF, M. A. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), ACM, pp. 470–478.

[79] KRČÁL, M., ŠVEC, O., BÁLEK, M., AND JAŠEK, O. Deep convolutional malware classifiers can learn from raw executables and labels only. In *ICLR Workshop* (2018).

[80] KRIZHEVSKY, A., HINTON, G., ET AL. Learning multiple layers of features from tiny images.

[81] KRUEGEL, C., KIRDA, E., MUTZ, D., ROBERTSON, W., AND VIGNA, G. Polymorphic worm detection using structural information of executables. In *International Workshop on Recent Advances in Intrusion Detection* (2005), Springer, pp. 207–226.

[82] KURAKIN, A., GOODFELLOW, I., AND BENGIO, S. Adversarial examples in the physical world. *arXiv preprint arXiv:1607.02533* (2016).

[83] LÁSZLÓ, T., AND KISS, Á. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30* (2009), 3–19.

[84] LAURENT, H., AND RIVEST, R. L. Constructing optimal binary decision trees is np-complete. *Information processing letters 5*, 1 (1976), 15–17.

[85] LECUN, Y., BOTTOU, L., BENGIO, Y., AND HAFFNER, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE 86*, 11 (1998), 2278–2324.

[86] LECUN, Y., JACKEL, L., BOTTOU, L., BRUNOT, A., CORTES, C., DENKER, J., DRUCKER, H., GUYON, I., MULLER, U., SACKINGER, E., ET AL. Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks* (1995), vol. 60, Perth, Australia, pp. 53–60.

[87] LI, J., LIU, Y., CHEN, T., XIAO, Z., LI, Z., AND WANG, J. Adversarial attacks and defenses on cyber–physical systems: A survey. *IEEE Internet of Things Journal 7*, 6 (2020), 5103–5115.

[88] LI, M. Q., FUNG, B. C. M., CHARLAND, P., AND DING, S. H. H. I-MAD: Interpretable malware detector using galaxy transformer. *Computers & Security* (2021), 102371.

[89] LI, X., LOH, P. K., AND TAN, F. Mechanisms of polymorphic and metamorphic viruses. In *Intelligence and Security Informatics Conference (EISIC), 2011 European* (2011), IEEE, pp. 149–154.

[90] LINARDATOS, P., PAPASTEFANOPOULOS, V., AND KOTSIANTIS, S. Explainable ai: A review of machine learning interpretability methods. *Entropy 23*, 1 (2020), 18.

[91] LINARDATOS, P., PAPASTEFANOPOULOS, V., AND KOTSIANTIS, S. Explainable ai: A review of machine learning interpretability methods. *Entropy 23*, 1 (2021), 18.

[92] LOH, W.-Y. Fifty years of classification and regression trees. *International Statistical Review 82*, 3 (2014), 329–348.

[93] LU, J., ISSARANON, T., AND FORSYTH, D. Safetynet: Detecting and rejecting adversarial examples robustly. In *Proceedings of the IEEE International Conference on Computer Vision* (2017), pp. 446–454.

[94] LUNDBERG, S. M., AND LEE, S.-I. A unified approach to interpreting model predictions. *Advances in neural information processing systems 30* (2017).

[95] MADRY, A., MAKELOV, A., SCHMIDT, L., TSIPRAS, D., AND VLADU, A. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083v4* (2019).

[96] MAHDAVIFAR, S., AND GHORBANI, A. A. Application of deep learning to cybersecurity: A survey. *Neurocomputing 347* (2019), 149–176.

[97] MILLER, D. J., XIANG, Z., AND KESIDIS, G. Adversarial learning targeting deep neural network classification: A comprehensive review of defenses against attacks. *Proceedings of the IEEE 108*, 3 (2020), 402–433.

[98] MOLNAR, C. *Interpretable machine learning*. Lulu.com, 2020.

[99] MOODY, R. Screen Time Statistics: Average Screen Time in US vs. the rest of the world. https://www.comparitech.com/tv-streaming/screen-time-statistics/#:~:text=The%20average%20American%20spends%20over,at%20a%20screen%20every%20day., 2022. [Online; accessed 7-May-202].

[100] MOOSAVI-DEZFOOLI, S.-M., FAWZI, A., AND FROSSARD, P. Deepfool: a simple and accurate method to fool deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016), pp. 2574–2582.

[101] MOSKOVITCH, R., FEHER, C., TZACHAR, N., BERGER, E., GITELMAN, M., DOLEV, S., AND ELOVICI, Y. Unknown malcode detection using opcode representation. In *Intelligence and Security Informatics*. Springer, 2008, pp. 204–215.

[102] MOURTAJI, Y., BOUHORMA, M., AND ALGHAZZAWI, D. Intelligent framework for malware detection with convolutional neural network. In *Proceedings of the 2nd International Conference on Networking, Information Systems & Security* (2019), ACM, p. 7.

[103] MUSTAFA, A., KHAN, S., HAYAT, M., GOECKE, R., SHEN, J., AND SHAO, L. Adversarial defense by restricting the hidden space of deep neural networks. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2019), pp. 3385–3394.

[104] MUSTAFA, A., KHAN, S. H., HAYAT, M., GOECKE, R., SHEN, J., AND SHAO, L. Deeply supervised discriminative learning for adversarial defense. *IEEE transactions on pattern analysis and machine intelligence 43*, 9 (2020), 3154–3166.

[105] NARODYTSKA, N., AND KASIVISWANATHAN, S. P. Simple black-box adversarial perturbations for deep networks. *arXiv preprint arXiv:1612.06299* (2016).

[106] NATARAJ, L., KARTHIKEYAN, S., JACOB, G., AND MANJUNATH, B. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security* (2011), ACM, p. 4.

[107] OLIVA, A., AND TORRALBA, A. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International journal of computer vision 42*, 3 (2001), 145–175.

[108] PAPERNOT, N., MCDANIEL, P., GOODFELLOW, I., JHA, S., CELIK, Z. B., AND SWAMI, A. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM Asia Conference on Computer and Communications Security (ASIA CCS)* (2017), pp. 506–519.

[109] PAPERNOT, N., MCDANIEL, P., JHA, S., FREDRIKSON, M., CELIK, Z. B., AND SWAMI, A. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on* (2016), IEEE, pp. 372–387.

[110] PAPERNOT, N., MCDANIEL, P., WU, X., JHA, S., AND SWAMI, A. Distillation as a defense to adversarial perturbations against deep neural networks. In *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P)* (2016), IEEE, pp. 582–597.

[111] PARK, S., AND SO, J. On the effectiveness of adversarial training in defending against adversarial example attacks for image classification. *Applied Sciences 10*, 22 (2020), 8079.

[112] PASZKE, A., GROSS, S., CHINTALA, S., CHANAN, G., YANG, E., DEVITO, Z., LIN, Z., DESMAISON, A., ANTIGA, L., AND LERER, A. Automatic differentiation in pytorch. In *NIPS Workshop* (2017).

[113] PEDREGOSA, F., VAROQUAUX, G., GRAMFORT, A., MICHEL, V., THIRION, B., GRISEL, O., BLONDEL, M., PRETTENHOFER, P., WEISS, R., DUBOURG, V., VANDERPLAS, J., PASSOS, A., COURNAPEAU, D., BRUCHER, M., PERROT, M., AND

DUCHESNAY, E. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research 12* (2011), 2825–2830.

[114] QUINLAN, J. R. Induction of decision trees. *Machine learning 1*, 1 (1986), 81–106.

[115] RADFORD, A., NARASIMHAN, K., SALIMANS, T., AND SUTSKEVER, I. Improving language understanding with unsupervised learning. Tech. rep., Technical report, OpenAI, 2018.

[116] RADFORD, A., WU, J., CHILD, R., LUAN, D., AMODEI, D., AND SUTSKEVER, I. Language models are unsupervised multitask learners. *OpenAI Blog 1* (2019), 8.

[117] RAFF, E., BARKER, J., SYLVESTER, J., BRANDON, R., CATANZARO, B., AND NICHOLAS, C. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435* (2017).

[118] RAFFEL, C., SHAZEER, N., ROBERTS, A., LEE, K., NARANG, S., MATENA, M., ZHOU, Y., LI, W., AND LIU, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv preprint arXiv:1910.10683* (2019).

[119] RAI, A. Explainable ai: From black box to glass box. *Journal of the Academy of Marketing Science 48*, 1 (2020), 137–141.

[120] RIBEIRO, M. T., SINGH, S., AND GUESTRIN, C. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining* (2016), pp. 1135–1144.

[121] ROYAL, P., HALPIN, M., DAGON, D., EDMONDS, R., AND LEE, W. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual* (2006), IEEE, pp. 289–300.

[122] RUDIN, C. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence 1*, 5 (2019), 206–215.

[123] RUSSELL, S. J., AND NORVIG, P. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[124] RUTKOWSKA, J. Redpill: Detect vmm using (almost) one cpu instruction. *http://invisiblethings. org/papers/redpill. html* (2004).

[125] SANTOS, I., DEVESA, J., BREZO, F., NIEVES, J., AND BRINGAS, P. G. Opem: A static-dynamic approach for machine-learning-based malware detection. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions* (2013), Springer, pp. 271–280.

[126] SAXE, J., AND BERLIN, K. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on* (2015), IEEE, pp. 11–20.

[127] SCHULTZ, M. G., ESKIN, E., ZADOK, F., AND STOLFO, S. J. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on* (2001), IEEE, pp. 38–49.

[128] SELVARAJU, R. R., COGSWELL, M., DAS, A., VEDANTAM, R., PARIKH, D., AND BATRA, D. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *Proceedings of the IEEE international conference on computer vision* (2017), pp. 618–626.

[129] SHAHAM, U., YAMADA, Y., AND NEGAHBAN, S. Understanding adversarial training: Increasing local stability of supervised models through robust optimization. *Neurocomputing 307* (2018), 195–204.

[130] SHRIKUMAR, A., GREENSIDE, P., AND KUNDAJE, A. Learning important features through propagating activation differences. In *International conference on machine learning* (2017), PMLR, pp. 3145–3153.

[131] SPRINGENBERG, J. T., DOSOVITSKIY, A., BROX, T., AND RIEDMILLER, M. Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806* (2014).

[132] SRIVASTAVA, R. K., GREFF, K., AND SCHMIDHUBER, J. Highway networks. *arXiv preprint arXiv:1505.00387* (2015).

[133] SUNDARARAJAN, M., TALY, A., AND YAN, Q. Axiomatic attribution for deep networks. In *International conference on machine learning* (2017), PMLR, pp. 3319–3328.

[134] SZEGEDY, C., ZAREMBA, W., SUTSKEVER, I., BRUNA, J., ERHAN, D., GOODFELLOW, I., AND FERGUS, R. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199* (2013).

[135] TU, C.-C., TING, P., CHEN, P.-Y., LIU, S., ZHANG, H., YI, J., HSIEH, C.-J., AND CHENG, S.-M. Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence* (2019), vol. 33, pp. 742–749.

[136] VASWANI, A., SHAZEER, N., PARMAR, N., USZKOREIT, J., JONES, L., GOMEZ, A. N., KAISER, Ł., AND POLOSUKHIN, I. Attention is all you need. In *Advances in Neural Information Processing Systems* (2017), pp. 5998–6008.

[137] WANG, D., LI, C., WEN, S., NEPAL, S., AND XIANG, Y. Defending against adversarial attack towards deep neural networks via collaborative multi-task training. *arXiv preprint arXiv:1803.05123* (2018).

[138] WILLE, R. Restructuring lattice theory: an approach based on hierarchies of concepts. In *Ordered sets*. Springer, 1982, pp. 445–470.

[139] WOLFE, P. Checking the calculation of gradients. *ACM Transactions on Mathematical Software (TOMS) 8*, 4 (1982), 337–343.

[140] XIAO, H., BIGGIO, B., NELSON, B., XIAO, H., ECKERT, C., AND ROLI, F. Support vector machines under adversarial label contamination. *Neurocomputing 160* (2015), 53–62.

[141] XIE, C., WU, Y., MAATEN, L. V. D., YUILLE, A. L., AND HE, K. Feature denoising for improving adversarial robustness. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (2019), pp. 501–509.

[142] XU, H., MA, Y., LIU, H., DEB, D., LIU, H., TANG, J., AND JAIN, A. Adversarial attacks and defenses in images, graphs and text: A review. *arXiv preprint arXiv:1909.08072* (2019).

[143] XU, W., EVANS, D., AND QI, Y. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).

[144] YAN, X., CHENG, H., HAN, J., AND YU, P. S. Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 433–444.

[145] YANG, Z., DAI, Z., YANG, Y., CARBONELL, J., SALAKHUTDINOV, R. R., AND LE, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)* (2019), pp. 5754–5764.

[146] YE, Y., LI, T., ADJEROH, D., AND IYENGAR, S. S. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR) 50*, 3 (2017), 41.

[147] YE, Y., LI, T., HUANG, K., JIANG, Q., AND CHEN, Y. Hierarchical associative classifier (hac) for malware detection from the large and imbalanced gray list. *Journal of Intelligent Information Systems 35*, 1 (2010), 1–20.

[148] YU, P., SONG, K., AND LU, J. Generating adversarial examples with conditional generative adversarial net. In *2018 24th International Conference on Pattern Recognition (ICPR)* (2018), IEEE, pp. 676–681.

[149] ZEILER, M. D., AND FERGUS, R. Visualizing and understanding convolutional networks. In *European conference on computer vision* (2014), Springer, pp. 818–833.

[150] ZHOU, B., KHOSLA, A., LAPEDRIZA, A., OLIVA, A., AND TORRALBA, A. Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 2921–2929.

[151] ZHOU, H., ZHANG, S., PENG, J., ZHANG, S., LI, J., XIONG, H., AND ZHANG, W. Informer: Beyond efficient transformer for long sequence time-series forecasting. In *Proceedings of AAAI* (2021).

[152] ZÜGNER, D., BORCHERT, O., AKBARNEJAD, A., AND GUENNEMANN, S. Adversarial attacks on graph neural networks: Perturbations and their patterns. *ACM Transactions on Knowledge Discovery from Data (TKDD) 14*, 5 (2020), 1–31.