NAGOYA UNIVERSITY

DOCTORAL THESIS

Effective Application of Artificial Intelligence Technology in Malware Detection and Classification

Author:
GAO Yun

Graduate School of Informatics

January 11, 2023

NAGOYA UNIVERSITY

Abstract

Graduate School of Informatics

Doctor of Informatics

Effective Application of Artificial Intelligence Technology in Malware Detection and Classification

by GAO Yun

Modern society relies more and more on computer system and network technology, and the threat of malicious software is becoming more and more serious. In the field of information security, malware detection has been a key problem that academia and industry are committed to solving. With the large-scale development of Artificial Intelligence (AI) technology, more and more information security personnel try to learn the feature of malware and normal software by machine learning, so that malware detection can get rid of threat intelligence and expert knowledge, and can calmly deal with large-scale malware attacks. This thesis proposes using different AI methods to detect and classify malicious software.

Traditional Machine Learning We investigated the malware detection results based on LightGBM in static malware detection methods, and reduced the false alarms by a custom log loss function, which controls the learning process of model through installing coefficient α to a loss function of the false-negative side and coefficient β to the false-positive side.

Graph Representation Learning We propose a Control-Flow Graph (CFG)- and Graph Isomorphism Network (GIN)-based malware classification system. The feature vectors of CFG basic blocks are generated using the large-scale pre-trained language model MiniLM, which is beneficial for the GIN to further learn and compress the CFG-based representation, and classified with multi-layer perceptron.

Graph Contrastive Learning We propose a malware classification framework based on Graph Contrastive Learning (GraphCL) with data augmentation. According to my experimental evaluation, the unsupervised learning approach outperformed the self-supervised learning approach in Graph Neural Networks based on malware classification.

This thesis has found that generally AI-based methods can effectively improve the detection and classification results of large-scale malware, and with the continuous improvement of AI technology, more and more AI technologies can be applied to the field of information security to help solve difficult security problems.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

1.1 Background

With the large-scale application of computer and network technology, the harm of malicious software has become particularly prominent. Network worms, ransomware, and other intrusion incidents are getting worse and worse. Hacker groups often use scripts to generate a large number of malware variants to carry out large-scale network attacks, resulting in heavy losses to all sectors of society.

According to the statistics of AV-TEST [1], From 2019 to 2020, more than 114 million malware reported by major security vendors were not recorded by the latest threat intelligence. In the first quarter of 2022, the AV-TEST systems have registered over 43 million newly-programmed samples. In today's network environment, malware attacks are complex and changeable. Traditional rule-based signature matching not only requires extensive expert knowledge to maintain the intelligence base, but is often helpless against emerging malware variants.

It is a possible way to solve this problem by applying Artificial Intelligence (AI) algorithm and detecting malware according to the behavior feature of malware. In fact, the application of machine learning to malware detection has been proposed as early as 1995 [2]. However, the training of a useful machine learning model needs to be supported by a sufficient number of high-quality data. Because there was no concept of big data at that time, there were few available samples of malware, so it is difficult for the model to learn the corresponding features effectively. In recent years, due to the rapid growth in the number of malware, researchers can collect large-scale malware samples, which provides a good data support for the training of machine learning and deep learning model. Besides statistical machine learning models, the vigorous development of deep learning algorithms such as Convolutional Neural Network (CNN), Recurrent Neural Network (RNN), and Graph Neural Network (GNN) also provides researchers with more choices. Under this background, more and more security researchers have designed feature engineering for malware detection, and applied various machine learning algorithms to detect malware, which finally achieved very good results. It is worth mentioning that machine learning method does not depend on expert knowledge and threat intelligence, and gives the evaluation result by learning the feature of malware, so it can discriminate variant malware at low cost.

1.2 Problem

Although machine learning has achieved remarkable results in malware detection and classification field, there are some problems due to the limitations of machine learning itself, which leads to the fact that machine learning is not widely used in industry.

First, the core problem is the robustness of machine learning model. Because machine learning is data-driven, the conclusions drawn on datasets often deviate from the real world. The detection results with extremely high accuracy are often caused by the model's over-fitting of data. Once such a model is put into production, it will become the target of black hat hackers' attack. Therefore, how to improve the generalization of model robustness is also a hot issue in academic circles.

Secondly, the interpretability problem of machine learning. As everyone knows, many machine learning models are designed according to the end-to-end model. Such a model is a black box for users and maintainers. People can't understand what decisions are made by the model, and what direction to optimize it. This makes machine learning often fail to give people absolute trust at the level of network security products.

Thirdly, the concept drift problem. As malware evolves over time, machine learning models trained through existing datasets may no longer be applicable at some point in the future, and retraining models are often accompanied by huge costs. Therefore, how to make the model last under the condition of low cost is also an urgent problem to be solved.

In addition, a challenging problem that arises in this domain is how to reduce false alarms. In many real-life business scenarios, including network security, false alarms are often more costly than missed alarms. If a malware detection system daily generates many false alarms, it places great pressure on incident-response staff. Their trust in the system eventually wanes, even though actual incidents might be overwhelmed with false alarms, leading to more and more serious false alarms.

1.3 Motivation

My research focuses on how to extract effective and interpretable malware features, apply the best detection model according to different features, continuously improve the detection accuracy and reduce the false alarm rate.

Because the structure and logic of malware are very complex, it is difficult to extract effective features. Traditional machine learning relies on feature engineering, and different features have great influence on the results of machine learning models. In addition, most data sets suitable for mainstream machine learning extract statistical characteristics of malware through surface analysis. Although the extraction speed of this method is faster, it ignores the structural information of rich call relationships in malware, such as CFG and call graph. Deep learning method does not need feature engineering, and end-to-end training is possible, but the detection result is often not as good as the traditional machine learning algorithm.

This research has the following values. First of all, starting with the custom loss function of machine learning algorithm, penalty parameters are added to the loss function. In the training stage of the model, the penalty for false alarm is increased, thus reducing the false alarm rate. Our custom loss function method frees users and security responders from massive false alarms. Secondly, by extracting the Control-Flow Graph (CFG) of malware, we get rich structural information, which includes the direct calling relationship of each malware basic block. Further using GNN, we get satisfactory detection results. The CFG of malware has a clear call relationship, which is itself natural graph data. With graph neural network, it can directly learn the structural feature of malware and the relationship between different graph nodes. Therefore, our malware detection model has good interpretability. Furthermore, supervised learning requires a lot of accurate labeling data. Because the number of newly added malware is too large, and there are many varieties of malware. It is unrealistic and inefficient to label all malicious software. By applying self-supervised Graph Contrastive Learning (GraphCL), this method can learn the graph structure feature by itself without labeling malicious software, so as to realize the detection and classification of malicious software. It has also achieved exciting results. Self-supervised learning does not need to label malicious samples, which greatly reduces the workload of dataset making. In addition, malware from different families have similar structural feature, and GNN is especially good at learning the graph structure information of malware. Therefore, our GraphCL model can classify unseen malware samples into families well, thus improving the concept drift problem to some extent.

1.4 Contribution

This thesis documents several key contributions made to the fields of malware detection and classification.

- First, we propose a custom log loss function that optimizes the malware classification model and substantially reduces false alarms. In addition, we propose a hybrid usage of different custom models to add additional priority to positive results that can reduce the workload of security response center persons. On a non-public dataset (FFRI dataset), malware detection can be effectively performed even with a small number of features.

- Secondly, we propose a malware detection system based on GNN, and kept the structure information of the samples extracted from CFG, and generated the text features of each node by a pre-trained language model. Furthermore, we created a special graph dataset for malware detection that can be directly used on GNN. As a new feature extraction method for malware, we compared the representation results of GNN with different dimensions under different classification models. The problem of malware detection is transformed into the problem of graph classification in the GNN field. The dimension of the feature vector extracted by the Graph Isomorphism Network (GIN) is very low (from 32 to 256 dimensions). As compared to the latest feature extraction method of the EMBER dataset (2381 dimensions), our model obtained very similar performance.

- Finally, we propose a malware classification framework based on graph contrastive learning under self-supervised learning, and retain the structural information of the samples extracted from CFG, and embed the text features of each node with a pre-trained language model. Furthermore, We create a special graph dataset for malware classification that can be used directly on GNN. Our pre-trained model can effectively perform a low-dimensional representation of malware with which a variety of downstream tasks can be performed. We have achieved good results on malware family classification tasks.

1.5 Thesis Outline

The remainder of this thesis has been organized into six chapters. This section outlines the description of each chapter:

- In Chapter 2, show in detail the state-of-the-art related work to malware detection and classification.
- In Chapter 3, introduce malware detection method using LightGBM-based custom logistic loss function.
- In Chapter 4, describe malware detection by CFG level representation learning with GIN.
- In Chapter 5, present self-supervised graph contrastive learning with data augmentation for malware classification.
- In Chapter 6, expose our final conclusion with future work.

Chapter 2

Related Work

Over the past few years, malware detection has evolved due to the gradual rising threat to large enterprises and government agencies posed by malware. Data mining and machine learning algorithms have been used extensively for malware detection, which is the process of analyzing the content of a program to determine whether it is malicious or benign. This chapter describes the related work in the field of malware detection and classification.

2.1 Static Malware Analysis

Malware is a blanket term for viruses, worms, trojans, and other harmful computer programs used by black hat hackers to cause destruction and gain access to sensitive information: "Malicious code is any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the system's intended function" [3]. The attacker's goal is to execute the malware in the user's environment and collect credit card data, usernames, passwords, and so on. Malware samples are analyzed to extract useful features that can be utilized to detect them in the future.

In static malware analysis techniques, malware samples are extracted as features without being executed. After static analysis, static features are extracted, including hashes, N-grams, opcodes, strings, and PE headers. I can exploit such features to design malware detection software (antivirus software, Intrusion Detection Systems, and so on.) [4]. During static malware analysis, I need to reverse engineer the malware sample to understand its internal structure and program code. The detection of malware samples from the code level requires converting executable malware files into assembly language code. Some of the most widely used debuggers and disassemblers, such as OllyDbg, IDA Pro, and WinDbg, are used to convert binary files into assembly code [5], [6]. By analyzing the disassembly code, a file's execution flow, structure, or pattern of malicious activities can be identified and used to detect new or existing malware variants. Studying assembly code to find execution patterns and features is complicated and time consuming. In addition, code obfuscation and packing techniques increase the difficulty of the analyst's job. Malware developers use such obfuscation techniques as code encryption, program instruction reordering, and dead code insertion techniques to evade malware analysis [7], [8]. Although I can easily obtain the disassembly code for unpacked malware, some must be unpacked first to get the complete sample structure and disassembly code.

The features of malware classification are sometimes extracted using data mining techniques. Data mining extracts new and meaningful information from large datasets or databases that were previously unknown. In recent years, new models and datasets have been created through data mining techniques [9]. The mainstream approach to extract features from PE-formatted malware under the Windows platform is surface analysis, such as using binary parsing tools like Library to Instrument Executable Formats (LIEF) to extract malware features and save them in the JavaScript Object Notation (JSON) file format. Researchers then need to extract features and convert the feature vectors themselves. Surface analysis uses specialized PE-parsing tools to extract features. Many other methods directly extract features from PE files based on Natural Language Processing (NLP) and image-based techniques. NLP-based feature extraction uses language models, such as N-gram models, to extract sample features, and image-based feature extraction methods convert malware features into gray-scale images [10], and so on.

2.2 Dynamic Malware Analysis

In dynamic analysis, malware needs to be executed first, and its activity is captured when the malware is running. The runtime behaviors include file system operations, registry key changes, process execution, and network activities [11]. Dynamic analysis is based on capturing the interaction between malicious software and computer system to identify malicious software. To prevent malicious software from damaging the host system, I need to capture its behavior in a controllable virtual machine. I can use conventional virtualization tools such as Virtual Box or VMware or automated malware analysis systems like Cuckoo Sandbox. When malware runs in a monitored environment, various activities are observed, such as creating new files, deleting system or user files, registry key changes, accessing URLs, API calls, downloading malware, or sending data to command and control systems. Depending on these activities, files are detected as either benign or malicious. Dynamic analysis techniques can analyze samples that are not detected correctly by static analysis.

2.3 Malware Feature Extraction

Although the problem of malware detection involves a variety of file formats and operating systems, in most cases, the same feature engineering and methods are also applicable to malware on other system platforms. For example, the feature extraction method applicable to windows system Portable Executable (PE) files is also applicable to malicious PDF files and malware on Linux or Android platforms. The following subsections will outline the representative methods of malware feature extraction.

2.3.1 Surface Analysis Feature

LIEF[12], a widely used feature extraction method, which extracts multidimensional features according to the related information of parsing the header and section of the binary file, including file byte-code features, import table information, entropy values of each part of the file, and so on. Those features together form a feature vector as a portrait of the detected file. which together form a feature vector as a portrait of the detected file. Based on this feature extraction method, researchers mostly use some statistical machine learning models, such as decision tree, support vector machine and ensemble learning, to conduct supervised training, and finally achieve the purpose of distinguishing benign software from malicious software.

2.3.2 Control-Flow Graph

The Control-Flow Graph (CFG) is a well-known graph-based program structure notation in the field of computer science. The program is separated with branch instruction (including branch instruction for function call use) and the separated blocks between branch instruction are called basic blocks. A typical end of the basic block is connected to one basic block or a plurality of basic blocks. Therefore, I can use the connecting notation of basic blocks to express the program structure; this connection graph becomes the CFG. The CFG is the basis of many compiler optimizations and static-analysis tools.

CFG is a well-known feature. Researchers convert binary software samples into assembly code through disassembly, and extract CFG from it, thus turning it into a graph classification problem. In recent years, the vigorous development of Graph Neural Network (GNN) has provided a new way of thinking to solve such problems. Representative work [13] and [14] both use graph neural network to analyze CFG, and the accuracy rate can reach more than 95% in large-scale datasets.

2.3.3 Network Traffic

Discovering malware network behavior through network traffic is a new malware detection technology in recent years, and some preliminary research results have been obtained. Detecting malware through network traffic does not require users to install detection programs on terminal devices, which greatly reduces the computing resources of user terminal devices.

With the full popularity of HTTPS, in order to ensure the security and privacy of communication, more and more network traffic is encrypted by HTTPS. What method is used to detect malicious traffic in encrypted traffic is very important, among which feature are the key of analysis. According to the path of traffic generation, from source to destination, from data generation, encapsulation to traffic transmission, it involves many feature, such as packet size, direction, protocol, traffic classification (service, application), and so on.

2.4 Large-scale Malware Detection and Classification

Static malware detection allows a sample to be classified as malicious or benign without having to execute it. On the contrary, dynamic malware detection is based on runtime behavior, including time-dependent system call sequences [15]–[17].

Although static detection is usually uncertain [18], it is obviously superior to dynamic detection, as dynamic detection identifies malicious files by executing the sample. Since 1995, various static PE malware detection methods based on machine learning have been proposed [2], [19]–[22].

2.4.1 Supervised Learning

Datasets Most researchers collect malicious and benign samples by themselves. Feature extraction is carried out by specific methods, and these features are used to make data sets for machine learning. Without a unified feature extraction method and dataset segmentation, it was difficult to compare different methods with other researchers in the security field. Furthermore, due to the specificity of the security field, malware-related datasets are less publicly available. In recent years, some malware detection datasets have been accepted and widely used in academic research, such as the EMBER dataset [23], the SOREL-20M dataset [24], and the BODMAS dataset [25]. Malware detection researchers tend to extract static features to construct detection systems. In addition, most existing dataset formats are unsuitable for data mining and machine learning algorithms. However, the original data represented by the EMBER dataset does not require researchers to extract features. This dataset directly provides feature vectors, which facilitates further model construction. EMBER dataset also come with baseline algorithms, which are the current state-of-the-art in malware detection. These baseline algorithms have basically achieved detection rates of over 90%. Specifically, EMBER's LightGBM baseline at a 0.1% False Positive Rate (FPR), the detection rate exceeds 93%, and at a 1% FPR, the detection rate exceeds 98%.

Machine Learning-based Methods Schultz et al. represented PE files by including such features as import functions, strings, and byte sequences [19]. Kolter et al. used techniques for byte-level N-grams and NLP, including Term Frequency-Inverse Document Frequency (TF-IDF) weighting of strings, to detect and classify malware [20]. Shafiq et al. proposed using just seven features from PE headers, motivated by the fact that most malware samples in their study typically exhibited those elements [26].

Decision Tree-based Methods The EMBER and SOREL-20M datasets both use the Light Gradient Boosting Machine (LightGBM) model as the baseline, which provides excellent malware detection results.

Decision Tree In machine learning, a decision tree is a predictive model that represents a mapping between an object's attributes and its values. Each node in the tree represents an object, each forked path represents a possible attribute value, and each leaf node corresponds to the value of the object represented by the path from the root node to the leaf node.

Gradient Boosting The fundamental difference between boosting and bagging is that the base model is not uniformly treated but is constantly tested and filtered to select the "elite," which are then given more votes, while the poor base models are given fewer votes. The final results are obtained by combining all the votes. In most cases, the boosting results are less biased. Boosting is a sequential process, where each subsequent model tries to correct the errors of the previous model. Therefore the succeeding models are dependent on the earlier models, and I need to train the models in sequence instead of parallel. Gradient Boosting implements boosting; the main idea is that each time a model is built, it is in the direction of a gradient decrease in the loss function of the previously built model.

Gradient Boosting Decision Trees A Gradient Boosting Decision Tree (GBDT) is an important integration learning algorithm that was deemed to be an algorithm with high generalization ability when it was first proposed. GDBT is an iterative decision tree algorithm based on boosting integration learning. Each iteration creates a new decision tree in the gradient of the reduced residuals and as many iterations as it takes to generate a decision tree.

Let $\{x_i, y_i\}_{i=1}^{n}$ denote a sample dataset. The basic learner is $h(x)$, where $x_i = (x_{1i}, x_{2i}, \ldots, x_{pi})$. $y_i$is the predicted label. L denotes the loss function. The following are the steps of GBDT [27] [28]:

Step 1: The following is the initial constant value of model δ:

$$F_0(x) = \arg\min_{\delta} \sum_{i=1}^{N} L(y_i, \delta). \tag{2.1}$$

Step 2: Calculate the value of the negative gradient of the loss function in the current model and use it as an estimate of the residuals. For number of iterations $m = \{1, 2, \ldots, M\}$,

$$y_i^* = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)-F_{\mathrm{m}-1(x)}}, \quad i = \{1, 2, \ldots, N\}. \tag{2.2}$$

Step 3: Fit the sample data and get initial model $h(x_i; \theta)$. By using the least square method, parameters θ_m of the model are obtained:

$$\theta_m = \arg\min_{\theta, \delta} \sum_{i=1}^{N} [y_i^* - \delta h(x_i; \theta)]^2. \tag{2.3}$$

Step 4: By minimizing the loss function, the current model weight is expressed:

$$\delta_m = \arg\min_{\theta,\delta} \sum_{i=1}^{N} L\left(y_i, F_{m-1}(x) + \delta h\left(x_i;\theta\right)\right). \tag{2.4}$$

Step 5: The model is updated:

$$F_m(x) = F_{m-1}(x) + \delta_m h\left(x_i;\theta_m\right). \tag{2.5}$$

This loop is executed until the specified number of iterations or convergence conditions are met.

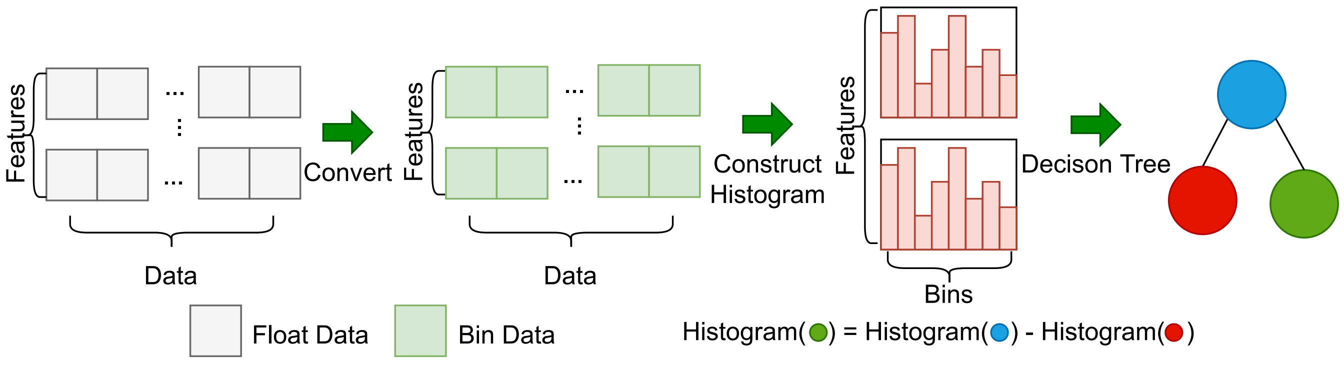


FIGURE 2.1: Histogram-based decision tree algorithm.

LightGBM LightGBM was designed by Microsoft [29] using a GBDT framework. The limitation of GBDT is that all the training data need to be traversed multiple times per iteration. If all the training data are loaded into the memory, the size of the training data will be limited. If not, repeatedly reading and writing the training data will consume much time. LightGBM was created to solve the inefficiency of GBDT when encountering large amounts of data. In LightGBM, the histogram-based algorithm and the trees' leaf-wise growth strategy with a maximum depth limit are adopted to increase the training speed and reduce memory consumption. The histogram-based decision tree algorithm is shown in Fig. 2.1.

LightGBM proposes two novel techniques: Gradient-based One-Side Sampling (GOSS) and Exclusive Feature Bundling (EFB). GOSS excludes a significant proportion of data instances with small gradients and only uses the rest to estimate the information gain. EFB bundles mutually exclusive features (i.e., they rarely take nonzero values simultaneously) such that it reduces the number of features.

Deep Learning-based Methods Saxe et al. used histograms through byte-entropy values as input features and multi-layer neural networks for classification [21]. Raff et al. showed that fully connected and recursive networks can be applied to malware detection problems [30]. They also used the raw bytes of PE files and set up end-to-end deep learning networks [22].

Chen et al. proposed robust PDF malware classifiers with verifiable robustness properties [31]. Coull et al. explored malware detection byte-based deep neural network models to learn more about malware and examined the learned features at multiple levels, from individual byte embeddings to end-to-end analysis of the models [32]. Rudd proposed Auxiliary Loss Optimization for Hypothesis Augmentation (ALOHA), which uses multiple additional optimization objectives to enhance the model, including multi-source malicious/benign loss, count loss on multi-source detections, and semantic malware attribute tag loss [33].

Deep Graph Learning-based Methods Graph classification assigns a label to each graph so that it can be mapped to the vector space. The graph kernel is dominant in history. It uses the kernel function to measure the similarity between graph pairs and uses certain mapping functions to map graphs to the vector space. In the background of graph classification, GNNs usually use readout operations to obtain a compact representation at the graph level. GNNs have attracted much attention and have demonstrated amazing results in graph classification tasks.

The Dynamic Graph Convolutional Neural Network (DGCNN) [34] uses K-Nearest Neighbors (KNN), builds a subgraph for each node based on the node's features, and then applies the graph convolution to the reconstructed graph. The Graph Isomorphism Network (GIN) [35] presents a graph isomorphism network that adjusts the weight of the central node through learning, theoretically analyzes the expression ability, is superior to GNN structures such as the Graph Convolution Network (GCN), and achieves state-of-the-art accuracy on multiple tasks.

2.4.2 Self-Supervised Learning

Unsupervised graph representation learning has already made good progress. For example, graph2vec [36] uses the set of all rooted subgraphs around each node as its vocabulary through a skip-gram training process.

Recently, contrastive learning has received much attention. It has also been applied in the field of malware detection and classification. Infograph [37] applies contrastive learning to graph learning, which is carried out in an unsupervised manner by maximizing the mutual information between graph-level and node-level representations. Yang presented a novel system called CADE, which can detect drifting samples that deviate from existing classes, and explained the detected drift [38]. EVOLIoT [39] is a novel approach that combats “concept drift" and the limitations of inter-family IoT malware classification by detecting drifting IoT malware families and examining their diverse evolutionary trajectories. This robust and effective contrastive method learns and compares semantically meaningful representations of IoT malware binaries and codes without expensive target labels. I proposed using self-supervised Graph Contrastive Learning with data augmentation for malware classification[40], the details are in section 5.

Chapter 3

Custom Loss Function of GBDT for Malware Detection

3.1 Introduction

The feverish development of machine learning and deep-learning-based artificial intelligence (AI) has resulted in such significant advances as image recognition and text sentiment analysis. Many cybersecurity applications have also been developed that use AI for security measures and protection from attacks. Malware software damages a single computer, server, or computer network. One malware and its variants can cause millions of dollars in damage, i.e., WannaCry with remotely exploitable EternalBlue, caused worldwide disasters, including the shutdown of the Japanese Honda Motor Company. Even though malware is becoming more sophisticated and diverse to avoid malware detection schemes, such schemes remain an essential issue in cybersecurity, especially as more and more people worldwide become dependent on computing systems.

Malware detection methods can be divided into static and dynamic malware detection [41]. Static methods classify samples as malicious or benign without executing samples; dynamic methods detect malicious software according to its runtime behavior. In theory, dynamic malware detection allows for the direct observation of malware actions that are not easily obfuscated and complicates reusing existing malware [42]. However, it is challenging to collect datasets of malware behavior because malware can identify sandbox environments and avoid executing its malicious actions. Moreover, dynamic malware detection in a practical environment requires many sandboxes to treat a plethora of doubtful samples, which increases detection costs. In contrast, while static malware detection is generally undecidable [18], massive datasets can be created by aggregating binary files and identifying malware before it is executed. Thus, this chapter proposes an idea for the static malware detection side since the field of malware detection is its main focus. Many widely believe that machine learning techniques can improve malware detection. As software technology advances and the internet evolves, thousands of pieces of malware are created every day. Such a massive flood of data poses a considerable challenge to malware analysts and Security Response Centers (SOCs). Over the past several decades, machine learning has played an important role in information security. A challenging problem that arises in this domain is how to reduce false alarms. In many real-life business scenarios, including network security, false alarms are often more costly than missed alarms. If a malware detection system daily generates many false alarms, it places great pressure on incident-response staff. Their trust in the system eventually wanes, even though actual incidents might be overwhelmed with false alarms, leading to more and more serious false alarms. In machine learning evaluation metrics, false alarms are equivalent to False Positives (FPs), which are caused by classification algorithms that incorrectly identify benign samples as malicious ones. Our goal is to reduce FPs as much as possible, even to zero. Malware detection is not a typical application of conventional machine learning, which focuses more on the balance between False Negatives (FNs) and FPs to achieve better Area Under the ROC Curve (AUC) metrics for machine learning classification. Reducing FPs is vital because even one FP among million benign samples can cause a range of consequences that might affect users. However, reducing FPs is complicated since thousands of benign samples are produced in the real world every day.

To the best of my knowledge, no previous research has investigated custom loss functions for malware detection purposes. Most existing malware detection research uses different feature engineering schemes and builds different machine learning models. My approach keeps the single-objective optimization model and controls the optimization objectives by further customizing the loss function, which can also achieve an improved model. I reduced the model's FP metric by customizing the loss function. Hence, I must put forward high requirements for the machine learning model and the optimization indicators during training and clearly focus on reducing the model's FPR. Therefore, during its training, I can reduce the FPR by customizing the loss function to give different weights to the FP. This chapter is an adapted version of the publication [43], [44] using GBDT with customized log loss function.

The specific contributions of this chapter can be summarized as follows:

- On a non-public dataset (FFRI dataset), malware detection can be effectively performed even with a small number of features.
- I propose a custom log loss function that optimizes the malware classification model and substantially reduces false alarms.
- I propose a hybrid usage of different custom models to add additional priority to positive results that can reduce the workload of security response center persons.

The remainder of this chapter is organized as follows. Section 3.2 reviews my proposed custom loss function and its application to malware detection. In Section 3.3, I discuss the corresponding experiments and evaluate their feasibility. In Section 3.4, I briefly discuss the results of the further analysis of different datasets and how to improve the SOC-response efficiency through our work. Finally, I describe the conclusion and future work in Section 3.5.

3.2 Proposed GBDT-based Custom Logistic Loss Function

3.2.1 Evaluation Environment

To evaluate our proposed custom log loss function, I constructed (for malware detection) different LightGBM-based cost-sensitive custom loss functions on two datasets, FFRI and EMBER. The evaluation environment of the proposed custom log loss function is shown in Fig. 3.1.

As shown in the figure, the preprocessing part of the two datasets I used are slightly different, mainly because the FFRI dataset does not directly provide feature vectors. Therefore I extracted and selected some valid features, while the EMBER dataset provides feature vectors directly, which can be easily used for subsequent experiments. Next I discuss the preprocessing details in Section 3.3.1.

3.2.2 Proposed Cost-sensitive Loss Function

In general, detection and classification algorithms of machine learning are only concerned with obtaining the highest accuracy rate. Regardless of the form of the loss function, the algorithm's prediction error formally consists of two components, FP and FN, and the loss function can be defined as follows:

$$Loss = Loss(FN) + Loss(FP). \tag{3.1}$$

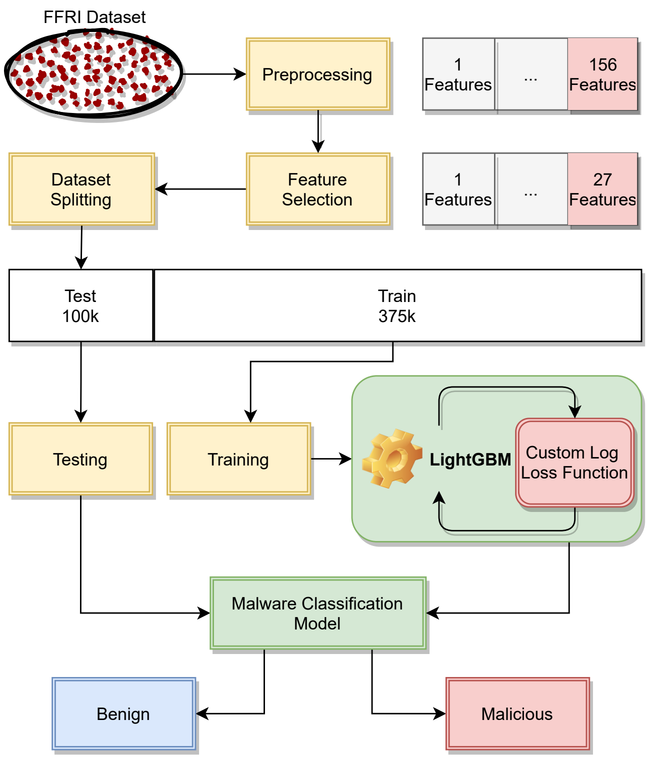
From the viewpoint of the general mathematical formulation of the loss function, the derivative optimization of the loss function is unbiased for FP and FN. The classification algorithm's concern is to make the value of $FP + FN$ as small as possible to achieve high classification accuracy. The traditional loss function does not consider any limitations on the respective weight of FP and FN in error instances.

Cross-Entropy Loss Cross-Entropy (CE), or log loss, measures the performance of a classification model whose output is a probability value between 0 and 1. The log loss increases as the predicted probability diverges from the actual label.

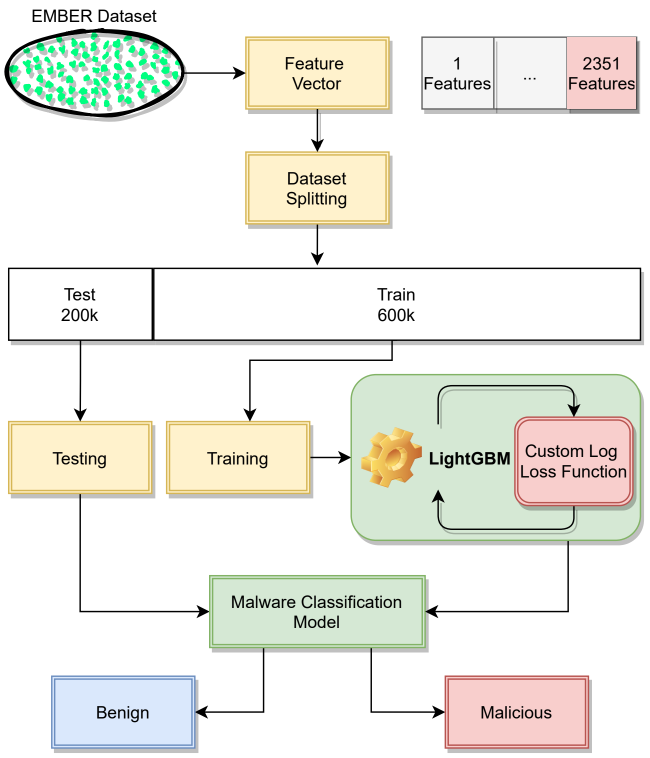
A perfect model has a log loss of 0. I introduce a custom log loss starting from the CE loss for binary classification:

$$\begin{aligned} CE(p, y) &= -y\log(p) - (1-y)\log(1-p) \\ &= \begin{cases} -\log(p), & \text{if } y = 1 \\ -\log(1-p), & \text{if } y = 0 \end{cases} \end{aligned} \tag{3.2}$$

In the above, y specifies the ground-truth class, and $p \in [0, 1]$ is the model's estimated probability for the class with label $y = 1$. For notational



(A) FFRI



(B) EMBER

FIGURE 3.1: Evaluation environment of proposed custom logistic loss function.

convenience, I define p_{t}:

$$p_{\mathrm{t}} = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{if } y = 0 \end{cases} \tag{3.3}$$

and rewrite $\mathrm{CE}(p, y) = -\log(p_{\mathrm{t}})$.

Cost-sensitive FN and FP Weighted Log Loss In malware detection, relative to the model's accuracy, since missing alarms (FN) and false alarms (FP) are more serious problems, we should be in the model training phase of false alarms and omissions for appropriate punishment. Our goal is to minimize the false alarm rate by penalizing the FN and FP that directly affect the FPR metrics. Based on this idea, we introduce two weighting factors: $\alpha \in (0, 500)$ for class 1 and $\beta \in (0, 500)$ for class -1. We set their values as an arithmetic progression between 0 and 500, with a common difference of 13. In practice, α and β can be set as hyperparameters through cross-validation. For notational convenience, we analogously define γ to how we defined p_{t} and write the custom CE loss as

$$\gamma = \begin{cases} \alpha & \text{if } y = 1 \\ \beta & \text{if } y = 0 \end{cases} \tag{3.4}$$

We write the CE loss as

$$CE\left(p_t\right) = -\gamma \log\left(p_t\right). \tag{3.5}$$

Finally, our custom log loss function can be defined as

$$\begin{aligned} L_{custom\_log\_loss} &= -\gamma \log(p_t) \\ &= \begin{cases} -\alpha \log(p), & \text{if } y = 1 \\ -\beta \log(1 - p), & \text{if } y = 0 \end{cases} \end{aligned} \tag{3.6}$$

I adopt this form in our experiments because it yields slightly improved accuracy than the standard log loss function. I explore suitable α and β values with the heuristic approach in one dataset. Then I validate the effectiveness of the obtained alpha and beta with another dataset to determine whether the custom log loss function with the accepted alpha and beta are effective among the datasets.

3.3 Experiment Setup

This section details the specifics of our experiment and results. Our experiments were conducted on a Manjaro 20.0.0 Lysia system (Linux kernel version is 5.6.16-1). The hardware specification is AMD Ryzen 7 3700X CPU and NVIDIA GeForce RTX 2070 SUPER GPU. The open-source libraries used in the experiments are Python 3.7.6, Anaconda 4.8.4, LightGBM 2.2.3, and Scikit-learn 0.22.2.

3.3.1 Datasets and Preprocessing

Datasets I used the FFRI dataset in our previous work [43]. In this study, I added a new dataset, EMBER, to validate the algorithm's effectiveness and to compare the experimental results of different datasets. I used EMBER dataset 2018 and FFRI dataset 2019 as our experimental datasets and show their details in Table 3.1. I can directly use the EMBER dataset, but on the FFRI dataset, I performed preprocessing to select useful features and to tune the model parameters.

TABLE 3.1: Our experimental datasets.

Dataset	Year	Train	Test	Features	Classes
EMBER	2018	600,000	200,000	2351	2
FFRI	2019	375,000	100,000	27	2

EMBER A cybersecurity company named Endgame[1] released a large open-source dataset called EMBER [23] in April 2018. EMBER collected over 1 million benign and malicious PE files. This dataset consists of JSON lines files, where each line contains a single JSON object. Each object includes the following fields: **sha256**, **appeared**, **label**, **general**, **header**, **imports**, **exports**, **section**, **histogram**, **byteentropy**, and **strings** fields. The EMBER dataset can be downloaded directly from GitHub[2].

FFRI The FFRI Dataset 2019, which was provided by the MWS Research Dataset [45], contains 250,000 malware samples and 250,000 cleanware samples: 500,000 pieces of data obtained from surface analysis. Each sample is a JSON file containing nine fields: **id**, **file_size**, **label**, **date**, **hashes**, **lief**, **peid**, **trid**, and **strings**. Each field contains multiple layers of JSON data. Since the FFRI dataset cannot be used directly, the MWS community must be contacted[3].

Preprocessing

PE Parser Both datasets leverage the Library to Instrument Executable Formats (LIEF) [12] as a PE parser. LIEF names are used for strings that represent symbolic objects, such as characteristics and properties. LIEF can efficiently extract all information directly from malicious and benign samples and store it in a JSON Lines format file.

EMBER The EMBER team has open-sourced their source code, which supports preprocessing, and the feature vectors are generated well. I do not

[1]https://www.endgame.com
[2]https://github.com/elastic/ember
[3]https://www.iwsec.org/mws/2019/about.html

go into the details of the methodology here; see the paper by the EMBER team. The dataset is available in two versions, 1 and 2. Our study uses all the features of version 1, which contains 2351 features. In the following subsection 3.3.2, I specifically discuss the features obtained from the EMBER dataset after preprocessing.

FFRI First, I read the fields needed from the FFRI Dataset 2019 JSON lines file and parsed the multi-layered JSON file structure layer by layer as preprocessing. Second, the parsed results are stored in a CSV file. I extracted 156 features and the labels of each sample for our dataset. Finally, I selected 75% from the dataset as a training set for a total of 375,000 pieces of data, and the test set used 50,000 bits of malware and 50,000 bits of cleanware, for a total of 100,000 data. Since our experiments use the LightGBM framework, no complex feature engineering is required. Numerical and categorical features can be used directly, although they need to be declared. In the following subsection 3.3.2, I discuss the features extracted from the FFRI dataset after preprocessing.

3.3.2 Features

EMBER Here, I briefly describe the feature groups, which are divided into two main categories: parsed features and format-agnostic features. Eight groups are shown in Table 3.2.

TABLE 3.2: Feature groups of EMBER dataset.

Feature group	Number of features
General file information (general)	10
Header information (header)	62
Imported functions (imports)	1,280
Exported functions (exports)	128
Section information (section)	255
Byte histogram (histogram)	256
Byte-entropy histogram (byteentropy)	256
String information (strings)	104
All	2,351

Parsed Features The dataset includes five groups of features extracted from the parsed PE files. Each parsed feature type is described in more detail below:

- **General file information (general)**: This type includes the file size, the virtual size, the number of imported and exported functions, whether the file has been debugged, the thread-local storage, the resources, the relocations, or signatures, and the number of symbols.

- **Header information (header)**: From the COFF header, a timestamp, a target machine, and a list of image characteristics are extracted. From the optional header, I extracted the target subsystem, the DLL characteristics, the file magic, the major/minor image versions, the linker versions, the system versions, the subsystem versions, the code, the headers, and the commit sizes.
- **Imported functions (imports)**: Pairs of dynamic link libraries and corresponding functions are formed, such as library:function name pairs (e.g., kernel32.dll:CreateFileMappingA).
- **Exported functions (exports)**: The exported functions are listed.
- **Section information (section)**: The entry point is the name of the section. The information of the section's properties includes the name, the size, the entropy, the virtual size, and a list of strings representing its characteristics.

Format-agnostic Features The dataset also includes three groups of format-agnostic features: a raw byte histogram, a byte-entropy histogram based on a previous work [21], and string extraction. These groups do not require that PE files be parsed for extraction:

- **Byte histogram (histogram)**: It contains 256 integer values that represent the count of each byte value in the file.
- **Byte-entropy histogram (byteentropy)**: It approximates the joint distribution of the entropy and byte values.
- **String information (strings)**: This dataset includes simple statistics information about printable strings. In particular, it reports the number of strings, their average length, a histogram of the printable characters in those strings, and the entropy of the characters in all the printable strings. In addition, the number of occurrences of a path, the URL, the registry key, and a short string MZ are also provided. Providing a simple statistical summary of the strings (instead of a list of the original strings) mitigates the privacy issues that might exist for some benign files.

FFRI In the FFRI dataset, eight groups of features are included with the corresponding labels. Four groups of features are inappropriate: **id**, **date**, **hashes** and **trid**. Finally, four groups are extracted from the FFRI datasets shown in Table 3.3.

Extracted Features Since the values of three groups (id, date, hashes) are unique, I did not use them in the experiment. I also discarded the **trid** features because their content is the probability of the type to which the detected file belongs, and the probability values of about five file types can be observed for a typical exe file. However, since these five file type identifiers

TABLE 3.3: Extracted feature groups of FFRI dataset.

Feature group	Number of features
File size information (file_size)	1
LIEF parsing information (lief)	32
PEiD parsing information (peid):	10
Raw string (strings):	100
All	143

are not uniform content, it is impossible to create a uniformly named feature type for each sample. In addition, the probability values of the judgment results of many samples are low, and constituting a valid feature is impossible, so I discarded the feature. I only used the **file_size**, **lief**, **peid**, and **strings** groups in our experiments. The features of these groups can be divided into numerical and category features:

- **Numerical features:** A numerical feature, which can be either continuous or discrete, is generally expressed as a real value. In general, decision tree type algorithms do not require any preprocessing of numeric features. As an example, in the lief.option_header.dll_characteristics field, 50.8% of the values are 0, 13.6% are 34112, and 19.3% are 320.

- **Categorical features:** A categorical feature indicates that a data point belongs to a certain class or has certain characteristics. LightGBM offers good accuracy with integer-encoded categorical features. As an example, in the peid.Anti-Debug field, 47% of the values are "no," 35% are "yes," and 19% are "no (yes)."

Selected Features I used the information from fields **file_size**, **lief**, **peid**, and **strings** as the features of our dataset and the **label** field as its label. I extracted 14 features from the lief field, including 11 numeric features and three categorical features, as well as five categorical features from the peid field. Furthermore, I extracted the first ten strings in the dataset, and the numbers that follow represent the order of the strings. Among these ten strings, I used seven features of high importance around the beginning because strings extracted around there are not too varied and are reasonable enough to be used as features. Through extensive experiments and feature importance ranking, I manually filtered a few of the most effective features to improve the model's training speed. The selected features of the FFRI dataset are shown in Table 3.4.

3.3.3 Models and Parameters

Our evaluation used two open-source libraries: Scikit-learn [4] version 0.22.2 and LightGBM [5] version 2.2.3. Microsoft open-sourced LightGBM in 2017.

[4]https://scikit-learn.org

[5]https://github.com/microsoft/LightGBM

TABLE 3.4: Selected features of FFRI dataset.

Feature names	Feature types (count)
file_size lief.header.characteristics lief.header.pointerto_symbol_table lief.header.time_date_stamp lief.header.numberof_sections lief.optional_header.imagebase lief.optional_header.checksum lief.optional_header.sizeof_initialized_data lief.optional_header.minor_linker_version lief.optional_header.dll_characteristics lief.entrypoint lief.virtual_size	numeric (12)
lief.sections lief.data_directories lief.optional_header.subsystem	categorical (3)
peid.PEiD peid.DLL peid.Packed peid.mutex peid.Anti-Debug	categorical (5)
strings_9 strings_7 strings_5 strings_8 strings_4 strings_10 strings_6	categorical (7)
All	numeric & categorical (27)

In the EMBER dataset, I used the default LightGBM parameters for training. The EMBER model parameters are shown in Table 3.5. Using GridSearchCV provided by the Scikit-learn library, it automatically tuned the parameters for the FFRI dataset model and obtained the optimal parameters. The default parameters have already obtained outstanding results. In order to get the best results, we used GridSearchCV to automatically tuned the parameters, although it was very time consuming. The FFRI model parameters are shown in Table 3.6.

3.3.4 Evaluation Metric

- **False Negative (FN)** denotes a binary classification error in which a test result incorrectly indicates a condition such as benign binary when the actual sample is malware.

TABLE 3.5: EMBER LightGBM baseline parameters.

Parameter name	Parameter value
boosting_type	gbdt
objective	binary
num_iterations	1000
learning_rate	0.05
num_leaves	2048
max_depth	15
min_data_in_leaf	50
feature_fraction	0.5

TABLE 3.6: FFRI LightGBM baseline parameters.

Parameter name	Parameter value
num_iterations	1000
random_seed	42
boosting_type	gbdt
objective	binary
metric	binary_logloss
learning_rate	0.1
num_leaves	123
colsample_bytree	0.8
subsample	0.9
max_depth	15
reg_alpha	0.1
reg_lambda	0.1
min_split_gain	0.01
min_child_weight	2
early_stopping_rounds	100

- **False Positive (FP)** is the opposite type of error where the test result incorrectly fails to indicate the presence of a condition when it is present.
- **True Positive Rate (TPR)** indicates the proportion of all the positive samples that are currently allocated to the true positive sample.
- **False Positive Rate (FPR)** indicates the proportion of true negative samples currently misclassified into the positive sample category out of the total number of negative samples.
- **Area Under the ROC Curve (AUC)**, is one of the most important for measuring the performance of binary classification model. It is a performance measurement for a classification problem at various thresholds settings. The ROC Curve measures how accurately the model can distinguish between two things. AUC measures the entire two-dimensional area underneath the ROC curve. This score gives us a good idea of how well the classifier will perform.

All evaluation metrics are shown in Table 3.7.

TABLE 3.7: Evaluation metrics.

Measure	Description
TP	Files correctly classified as malicious
TN	Files correctly classified as benign
FP	Files mistakenly classified as malicious
FN	Files mistakenly classified as benign
TP Rate (TPR)	$\frac{TP}{TP+FN}$
FP Rate (FPR)	$\frac{FP}{FP+TN}$
AUC	Area under ROC Curve

TABLE 3.8: Experimental results.

Dataset	Model	Parameter			Metrics				
		α	β	α / β	FN	FP	TPR	FPR	AUC
EMBER	SVM (LinearSVC)	-	-	-	33469	45457	0.66531	0.45457	0.60537
	Normal(BaseLine)	1	1	1	2833	1392	0.97167	0.01392	0.97887
	Custom_FN_FP	404	430	0.9395	**2803**	**1370**	0.97197	0.01370	0.97913
	Custom_TPR	222	1	222	184	60654	**0.99816**	0.60654	0.69581
	Custom_FPR	1	248	0.0040	7042	333	0.92958	**0.00333**	0.96312
	Custom_AUC	430	339	1.2684	2702	1462	0.97298	0.01462	**0.97918**
FFRI	SVM (LinearSVC)	-	-	-	16825	788	0.66350	0.01576	0.82387
	Normal (BaseLine)	1	1	1	227	118	0.99546	0.00236	0.99655
	Custom_FN_FP	404	417	0.9688	**215**	**110**	0.99570	0.00220	0.99675
	Custom_TPR	443	1	443	80	343	**0.99840**	0.00686	0.99577
	Custom_FPR	1	417	0.0024	479	34	0.99042	**0.00068**	0.99487
	Custom_AUC	469	287	1.6341	181	129	0.99638	0.00258	**0.99690**

3.3.5 Experimental Results

Our experiment used two datasets, and when the values of α and β are 1, it is our baseline model. To improve the description of our experimental results, I calculated the ratio of α and β (α/β). Compared with the two dimensions of alpha and beta, one-dimensional α/β produced better graphics for analysis. I counted the FN, FP, TPR, FPR, and AUC values for validation. The best results for datasets EMBER and FFRI are shown in Table 4.5. The classification threshold in the experiments is 0.5, which means that samples whose model prediction probability exceeds 0.5 are malicious and those less than 0.5 are benign.

I also separately trained the classification models using a traditional machine learning algorithm Linear Support Vector Classification (LinearSVC) to facilitate a comparison with our method. I used the default parameters and did no parameter tuning for a simple comparison. The Normal model (α=1 and β=1) is our main comparison baseline.

Finally, I got four optimal models on two datasets with our proposed method: Custom_FN_FP, Custom_TPR, Custom_FPR, and Custom_AUC. For FN and FP metrics, the model with the best results is Custom_FN_FP. Similarly, the model Custom_AUC with the best AUC result. Through repeated experiments, let α and β take values from 0 to 500, respectively. I found that with the EMBER dataset, when α=430 and β=339 (i.e., α/β=1.2684), the model's AUC is optimal. With the FFRI dataset, when α=469 and β=287 (i.e., α/β=1.6341), the model's AUC is optimal. The custom log loss model has a significant advantage for reducing the false alarm rates.

I usually consider the optimal model to be the one with the highest AUC metric, although due to the specificity of the information security domain, false positives are unacceptable. I discuss the optimal model further in Section 3.4.

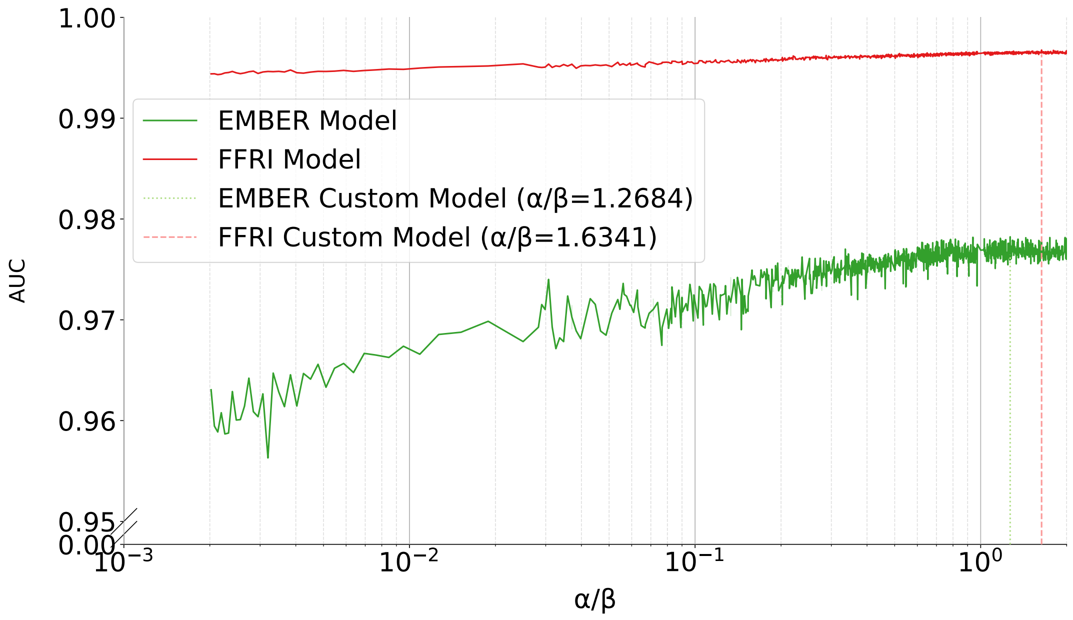
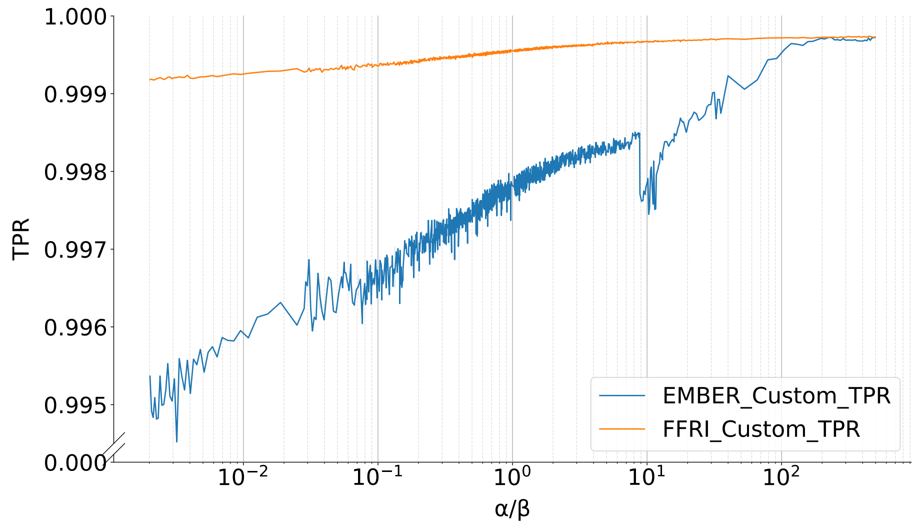


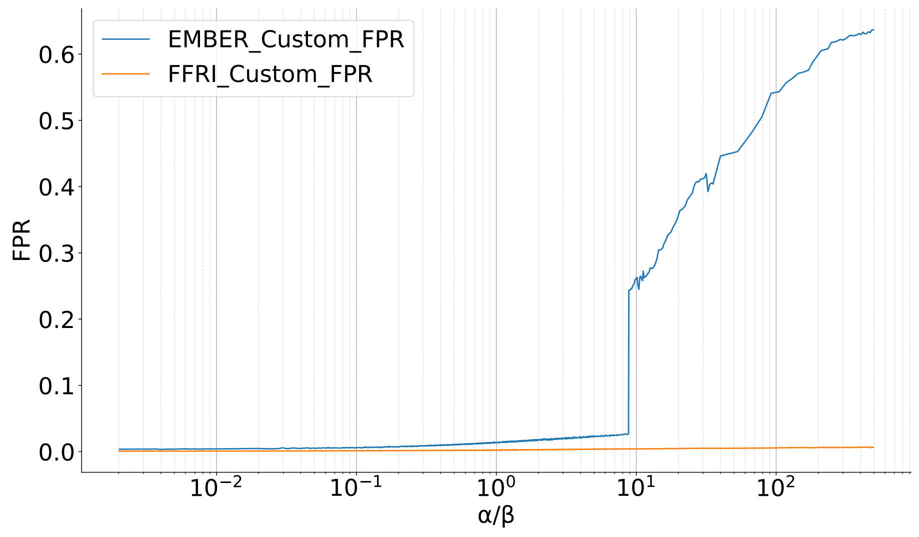
FIGURE 3.2: EMBER vs. FFRI AUC with logarithmic scale (α/β).

AUC Metrics Result Figure 3.2 compares the two models in the same coordinate system before scaling down the x-axis to obtain the optimal model. The x-axis denotes parameter α/β, and the y-axis represents the AUC scores. The AUC value of the EMBER model changes more with α/β than that of the FFRI model, especially when the value of α/β is close to 10, which is the point at which the AUC starts to drop sharply. In contrast, the overall change of AUC with the FFRI dataset is smooth, and the AUC value of both models is optimal when $\alpha/\beta \in [1,2]$.

There is a significant gap in the overall AUC between the EMBER and FFRI models, which might have been caused by the more effective features extracted from the FFRI dataset and the long-term model parameter tuning. In addition, the improvement of the FFRI model's AUC is smooth, and that of the EMBER model is great, showing that our method is more effective for the EMBER dataset.



(A) TPR with logarithmic scale (α/β).

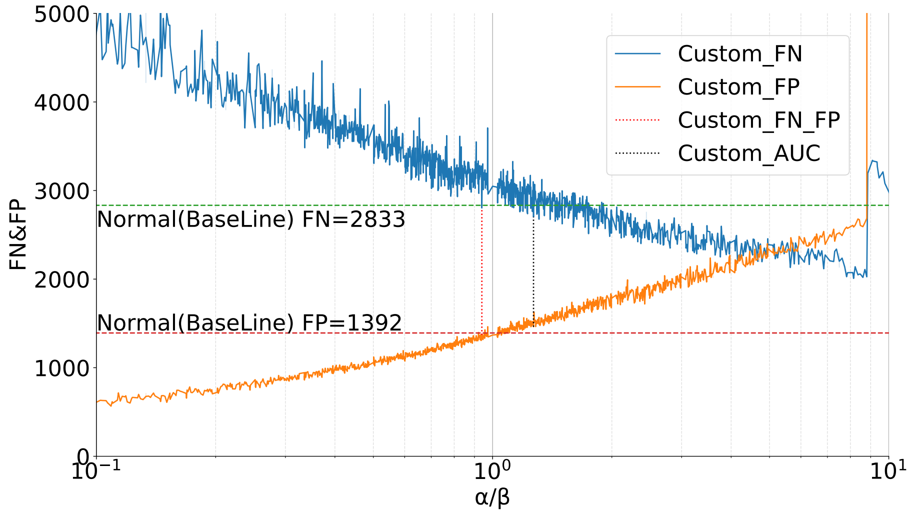


(B) FPR with logarithmic scale (α/β).

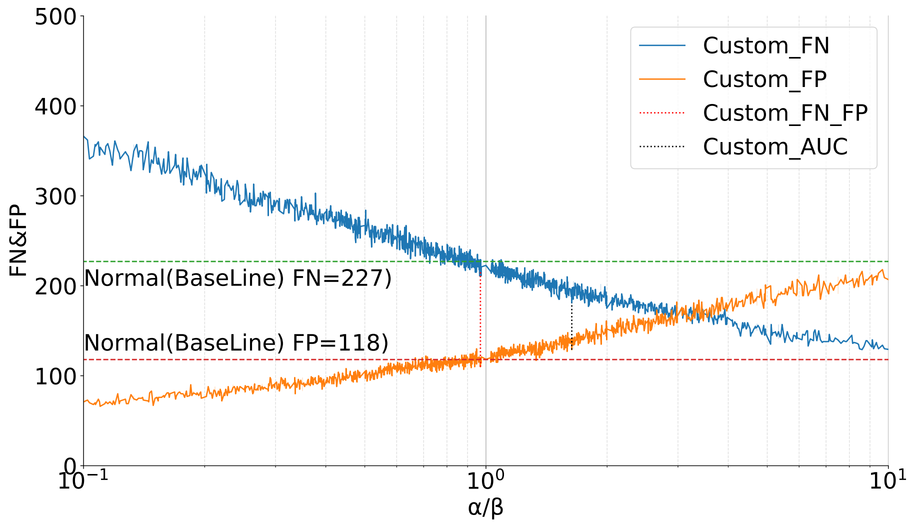
FIGURE 3.3: EMBER vs. FFRI TPR&FPR with logarithmic scale (α/β).

TPR and FPR Metrics Result Figure 3.3 compares TPR and FPR with α/β for the custom and normal models on the two datasets. The TPR results improved with increasing α/β values on both datasets. The enhancement of the custom model is much more significant in the EMBER dataset than in the FFRI. The FPR value increases with greater α/β. For the EMBER dataset, the FPR increases substantially when the value of α/β is close to 10; for the FFRI dataset, the FPR increases less.

FN vs. FP Metrics Result Figure 3.4 compares FN and FP with α/β for the custom and normal models on two datasets. Our custom model, Custom_FN_FP, is below the FN metric of the baseline model and the FP metric on both datasets. On the other hand, custom model Custom_AUC has a lower FN metric than the baseline model and a higher FP metric than the baseline model.



(A) EMBER



(B) FFRI

FIGURE 3.4: EMBER vs. FFRI FN&FP with logarithmic scale (α/β).

3.4 Discussion

From the experimental results of both datasets, the results on the FFRI dataset are always better than the EMBER results. I believe this result reflects that the features extracted by FFRI are more effective; another important reason is the smaller data size of FFRI relative to EMBER. According to Figure 3.5, the model has the highest AUC results when $\alpha/\beta \in [1, 2]$. When α/β tends to 0, that is, where α is 1 and β is as large as possible within a certain range, the model's FPR metric is optimal. For the optimal configuration of alpha and beta, an optimal classification model is the one with the highest aggregate AUC metric.

However, in the field of information security, I believe that I can sacrifice some model performance to minimize the false alarm rate. Therefore, for a special case in the malware detection field, the model with the lowest FPR is also the optimal model. When the lowest false alarm rate is studied, the optimal configuration on the EMBER dataset is $\alpha = 1$, $\beta = 248$. On the FFRI dataset, the optimal configuration is $\alpha = 1$, $\beta = 417$.

3.4.1 EMBER Dataset Results Analysis

AUC with Logarithmic Scale (α/β) Figure 3.5(A) compares AUC with α and β for the custom and normal models on the EMBER datasets. The EMBER model's AUC changes more with α/β than that of the FFRI model, especially when the value of α/β is close to 10, at which the AUC starts to drop sharply.

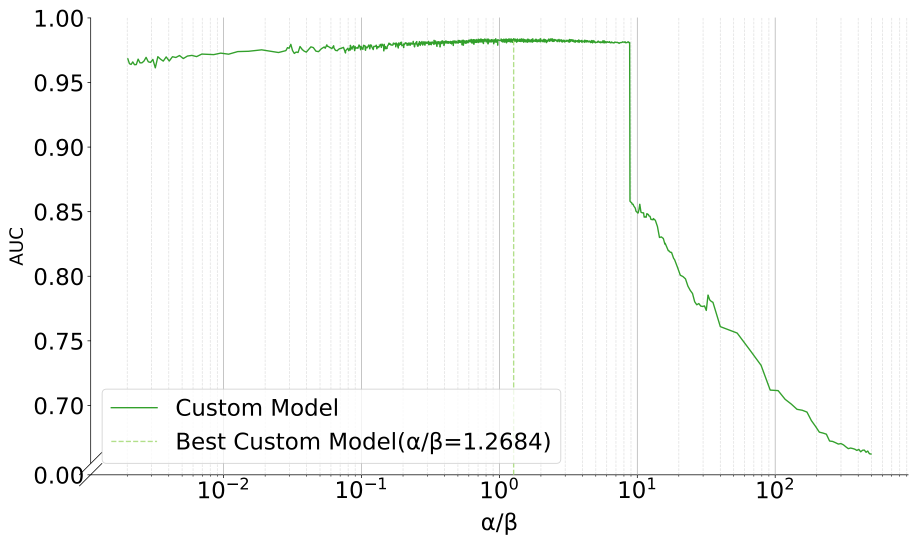
AUC Heatmap with α and β Figure 3.6(A) compares the AUC heat map with α and β for the custom and normal models on the EMBER dataset. I used a heat map to highlight the effect of α and β on the model's AUC. The distribution of AUC (especially in the high AUC results) and the variation trends are basically identical for both datasets, demonstrating the effectiveness of our method on different datasets.

The AUC of the Custom_AUC model for the EMBER dataset is located in the lower right region of the heat map. The model has the highest AUC on the EMBER dataset when α=430 and β=339.

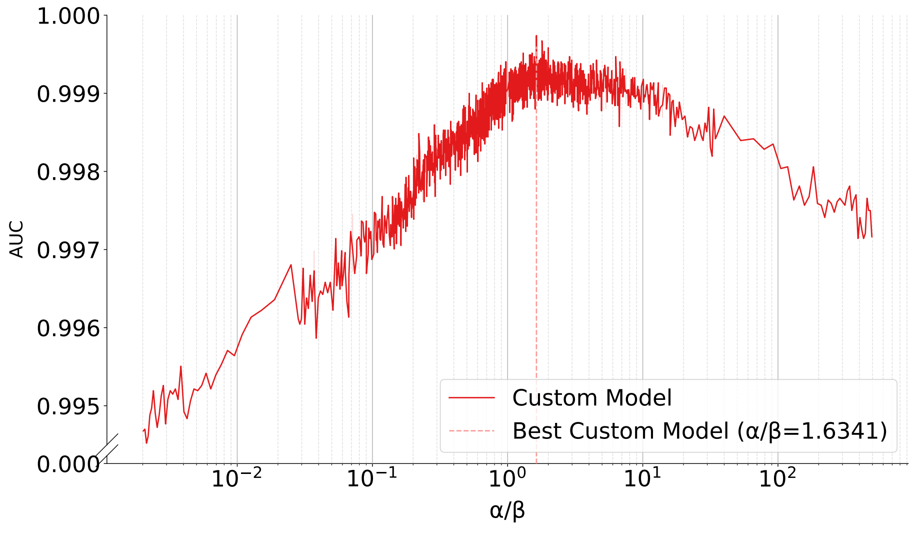
3.4.2 FFRI Dataset Results Analysis

AUC with Logarithmic Scale (α/β) Figure 3.5(B) compares the AUC with α and β for the custom and normal models on the FFRI datasets. In contrast, on the FFRI dataset the overall change of AUC is smooth, and the AUC value of both models is optimal when $\alpha/\beta \in [1, 2]$.

AUC Heatmap with α and β Figure 3.6(B) compares the AUC heat map with α and β for the custom and normal models on the FFRI dataset. The AUC of the Custom_AUC model for the FFRI dataset is located in the lower right region of the heat map. The model had the highest AUC when α=469 and β=287.



(A) EMBER AUC



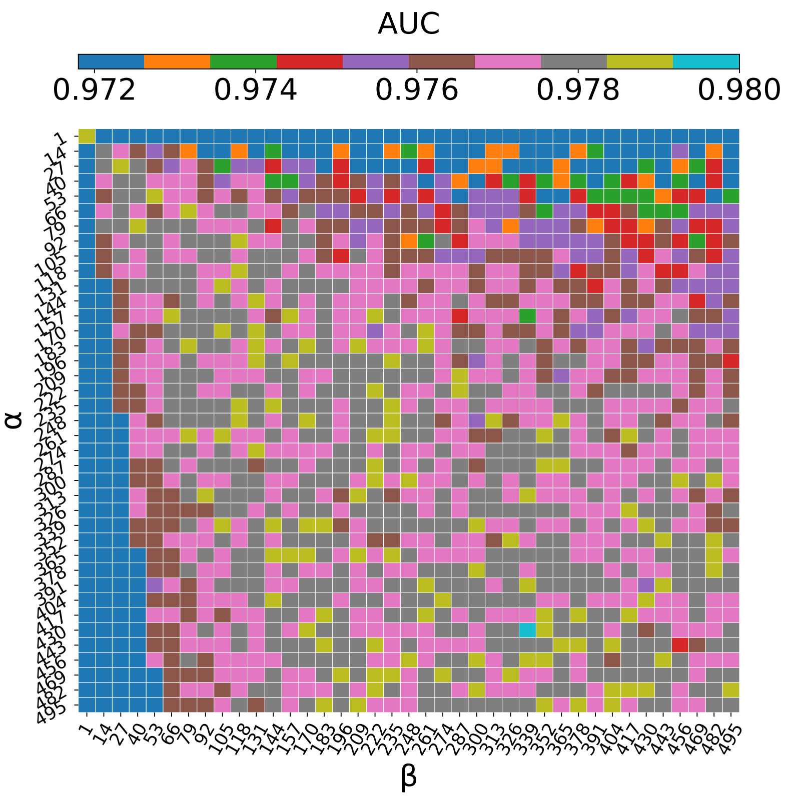
(B) FFRI AUC

FIGURE 3.5: EMBER vs. FFRI AUC with logarithmic scale (α/β).

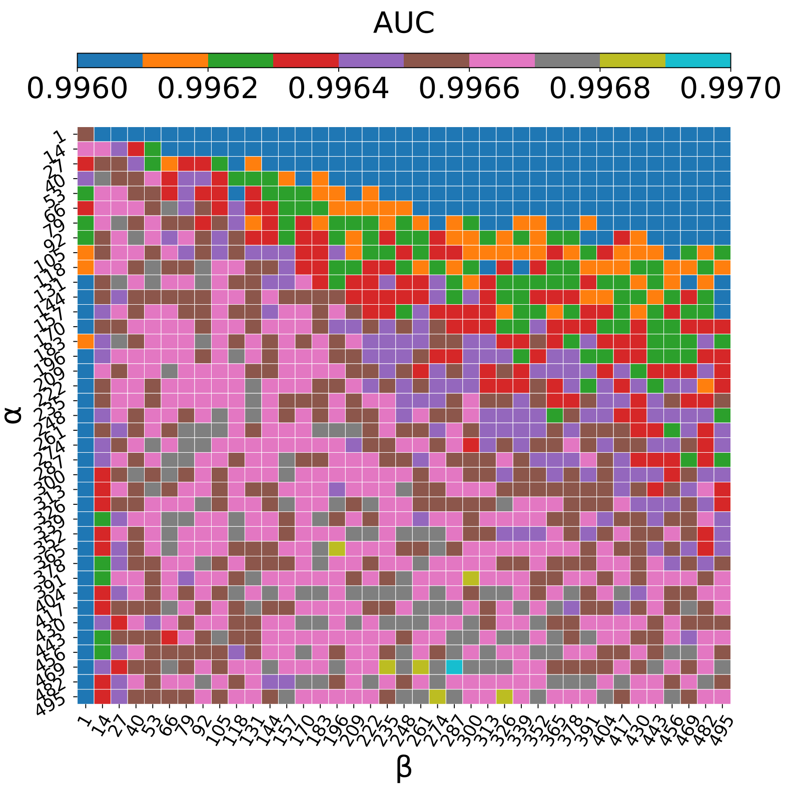
3.4.3 Hybrid Usage with Different Custom Models

Both models yield different detection results. By utilizing these two models to improve security operations, I can prioritize countermeasures for more positive results. One huge problem in security operations is that many FP results hide a few TP results and delay TP-incident countermeasures.

To achieve both quick countermeasures to actual TP results and to avoid hasty FN results, I propose a hybrid usage of both the Custom_FPR and Custom_AUC models. The proposal prioritizes the positive results in them (Table 3.9). If the Custom_FPR model gives a positive result, it may be a TP result in high accuracy, and so that this result becomes the 1st prioritized security incident for the security responders. However, the Custom_FPR models create more FN results than the Custom_AUC models. A positive result



(A) EMBER



(B) FFRI

FIGURE 3.6: EMBER vs. FFRI AUC heatmap.

TABLE 3.9: Combining Custom_AUC and Custom_FPR models to improve countermeasure priority.

	Custom_AUC model	Custom_FPR model
1st Priority	Positive/Negative	Positive
2nd Priority	Positive	Negative
3rd Priority	Negative	Negative

in Custom_AUC models but a negative result in Custom_FPR models becomes a 2nd priority, a security-incident candidate. The remainder becomes 3rd priority candidates and might not contain TP results. As shown in Table 3.8, even in the Custom_AUC model, there are 1462 pieces of FP data. However, the Custom_FPR model contains 333 pieces pf FP data: 23% of the Custom_AUC model. To initiate incident responses from the positive results in the Custom_FPR model, I overlook 1129 FP pieces of data in the result of the Custom_AUC model but not in the result of the Custom_FPR model.

Although the number of positives is small in our result, in the practical operations of Security Operations Centers, the number of binary samples that a classifier must recognize is very large, and so the number of positive samples increases and becomes comparatively large. SOC might require a few to several tens of minutes until it initiates a counteraction to a positive binary sample. Imagine an action to an actual positive sample has been delayed due to responding to false-positive samples. In such cases, larger damages (e.g., further malware dissemination) will occur. Thus, I believe that the ability to prioritize positive samples is very effective for practical SOC operation.

3.5 Conclusion

This study proposed a custom log loss function with α and β parameters to the LightGBM algorithm to solve the malware detection problem. I extracted 27 valid features from a non-public FFRI dataset for malware detection. I validated the effectiveness of our proposed method by separately and simultaneously evaluating it on FFRI and another public dataset, EMBER. Our result shows that the custom log loss function can reduce FP more than the normal log loss function. However, since the custom log loss function increased FN more than the normal log loss function, I also proposed a hybrid usage of the Custom_AUC and Custom_FPR models to prioritize the positive results. Reducing the FPR by the custom log loss function will significantly lower the priority given to false alarms, thus alleviating the pressure on security responders. Although I can control the model to learn more difficult samples by customizing the loss function, learning them is inherently difficult for the model. It is difficult to achieve complete learning; it inevitably leads to a certain number of FPs and FNs. Our proposed method penalizes FP and FN in the training phase and can only make them as low as possible instead of zero. I believe that the fundamental reason lies in the data themselves, the size of the dataset, and the effectiveness of the extracted features.

Chapter 4

Malware Control-Flow Graph Level Representation Learning

4.1 Introduction

With the progress of software technology and the development of the Internet, thousands of malware are produced every day [46] due to the proliferation of tools for creating and disguising malware[47]. Such a large amount of data creates great challenges for malware analysts and Security Operation Centers (SOCs). Traditional malware detection methods cannot quickly and effectively detect a large number of newly created malware. Machine learning is a promising method that can detect and classify large-scale newly generated malicious software according to the features of samples [48]. Over the past several decades, machine learning has played an important role in the information security domain, but the difficult problem of how to reduce false alarms remains.

The feature extraction method of Portable Executable (PE) files used in the Endgame Malware Benchmark for Research (EMBER) dataset [23] is widely used. In recent years, many publicly available datasets have focused on malware detection, including the Sophos-ReversingLabs 20 Million dataset (SOREL-20M) [24] and Blue Hexagon Open Dataset for Malware Analysis (BODMAS) [25], both of which use the same feature extraction method. This approach directly provides researchers with consistent feature vectors, allowing individuals in the same field to compare their respective methods. The datasets also come with baseline algorithms that are the current state-of-the-art in malware detection, with a detection rate that has exceeded 90%. These datasets all use the same feature extraction method.

Although this feature extraction method is very effective, there are still some new challenges, such as the high dimension of feature vectors and the inability to apply to Graph Neural Network (GNN). In addition, the interpretability of current malware feature extraction methods is poor, so a new detection method is needed to improve the interpretability of features and models. I propose a new feature extraction method of PE files based on the Control-Flow Graph (CFG) to solve these challenges. Information related to software structure (such as the CFG) is seldom used and most methods extract statistical information as features based on surface analysis. In recent years, GNNs have made significant progress [34]–[36] toward improved malicious software detection. The difficulty lies in how to represent malware in the form of graphics. The CFG is a natural graph structure and the graph structure data of malware can be generated by extracting the CFG.

Our research motivation is to reduce false alarms by using a new feature extraction method. In addition, the dataset obtained by traditional feature extraction methods can only be learned with a machine learning model, but cannot be applied to GNNs. Therefore, I try to detect malicious software by building a graph dataset and use this dataset to train a GNN to further detect and classify malware. Since there is no publicly available graph classification dataset for malware detection, I embarked on creating such a dataset.

This chapter is an adapted version of the publication [49], [50] using a Graph Isomorphism Network (GIN) for malware detection. As a supplement to our previous research, I added samples of several malware families to expand the dataset, used cross-validation to test the effectiveness of our method, and further tested the validity of the GNN representation using different classification models. In addition, I generated malware representations extracted by GNNs in different dimensions and tested the effects of different dimensions on the classification model results.

Our contributions can be summarized as follows:

- I proposed a malware detection system based on GNNs.
- I kept the structure information of the samples extracted from CFG and generated the text features of each node by a pre-trained language model.
- I created a special graph dataset for malware detection that can be directly used on GNNs.
- As a new feature extraction method for malware, I compared the representation results of GNNs with different dimensions under different classification models.
- The problem of malware detection is transformed into the problem of graph classification in the GNNs field. The dimension of the feature vector extracted by the GIN is very low (from 32 to 256 dimensions). As compared to the latest feature extraction method of the EMBER dataset (2381 dimensions), our model obtained very similar performance.

The remainder of this chapter is organized as follows. Section 4.2 reviews our proposed GIN-based static PE malware detection system and describe its application to malware detection. In Section 4.3, I briefly discuss the implementation details of our proposal. In Section4.4, I introduce the corresponding experiments and evaluate their feasibility. In Section 4.5, I discuss the limitations and advantages of our proposal. Finally, I describe our future work in Section 4.6.

4.2 Proposed GIN-based Static PE Malware Detection System

Our proposed malware detection system can directly extract CFG-based graph information of malware from PE files and then compress the CFG information into the feature vector with GIN. Ultimately, the system classifies feature vectors into malicious or benign ware with Multi-Layer Perceptron (MLP). I elaborate the system in detail.

4.2.1 Preliminary GNN Summary

I start by summarizing some of the most common GNN models and, along the way, introduce our notation. Let $G = (V, E)$ denote a graph with node feature vectors X_v for $v \in V$. $\mathcal{N}(v)$ is a set of nodes adjacent to v. On the graph classification task: given a set of graphs $\{G_1, \ldots, G_N\} \subseteq \mathcal{G}$ and their labels $\{y_1, \ldots, y_N\} \subseteq \mathcal{Y}$, I aim to learn a representation vector h_G that helps predict the label of an entire graph, $y_G = g(h_G)$.

GNNs follow the neighborhood aggregation strategy, which iteratively updates the node representation by aggregating representations of its neighbors. After k iterations of aggregation, the node representation captures the structural information within its $k - hop$ network neighborhood. Formally, the $k - th$ layer of a GNN is

$$a_v^{(k)} = \text{AGGREGATE}^{(k)} \left(\left\{ h_u^{(k-1)} | u \in \mathcal{N}(v) \right\} \right) \tag{4.1}$$

$$h_v^{(k)} = \text{COMBINE}^{(k)} \left(h_v^{(k-1)}, a_v^{(k)} \right), \tag{4.2}$$

where $h_v^{(k)}$ is the feature vector of node v at the $k - th$ iteration/layer. For a graph classification task, the READOUT function aggregates node features from the final iteration (K) to obtain the representation h_G of the entire graph:

$$h_G = \text{READOUT} \left(\left\{ h_v^{(K)} \mid v \in G \right\} \right). \tag{4.3}$$

K is the final iteration/layer.

4.2.2 Proposed System Architecture

The system consists of three parts: the Graph Feature Extraction (GFE) module that extracts the CFG-based feature from the PE files, the Graph Data Generation (GDG) module, and the Graph Classification (GC) module that compresses node feature vectors with GIN and classifies them with MLP. The overview of our proposed malware detection system is shown in Figure 4.1. I collected original PE files from public datasets as training samples of the malware detection system, including malicious samples and benign samples. These samples can generate a suitable malware dataset for GNNs through the GFE and GDG modules. Finally, I use the GC module to extract graph-level representation through a GNN and use this representation to train a binary classification MLP model to obtain malware detection results.

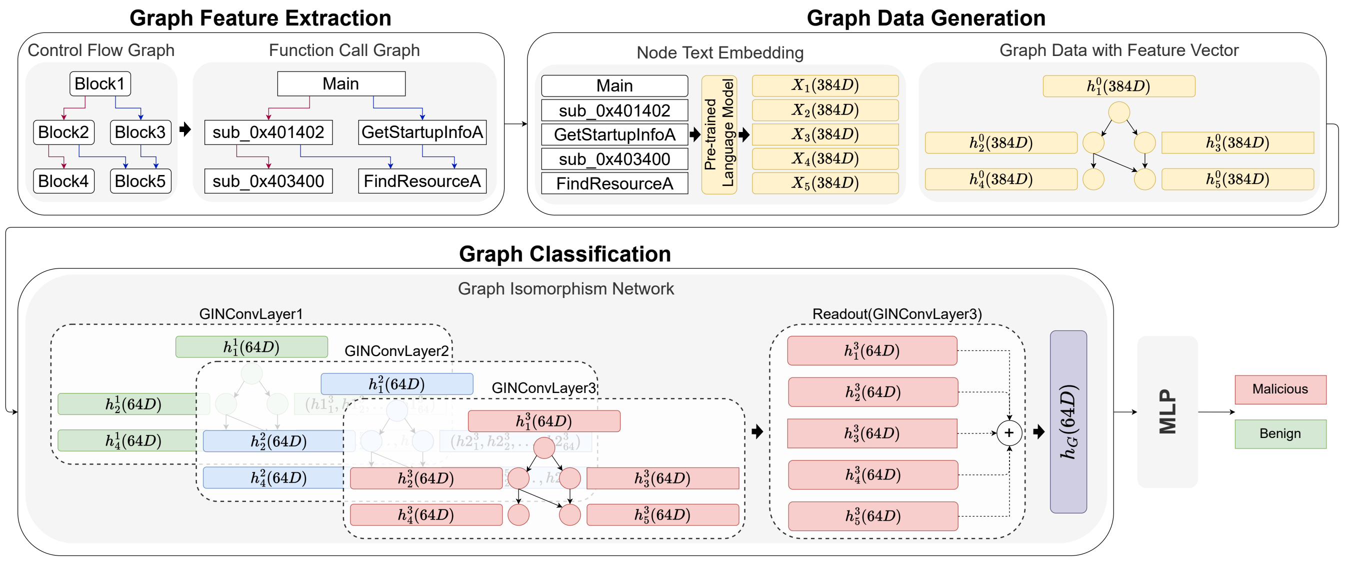


FIGURE 4.1: Proposed GIN-based static PE malware detection system.

Graph Feature Extraction Module The CFG is a well-known graph-based program structure notation in the field of computer science. The program is separated with branch instruction (including branch instruction for function call use) and the separated blocks between branch instruction are called basic blocks. A typical end of the basic block is connected to one basic block or a plurality of basic blocks. Therefore, I can use the connecting notation of basic blocks to express the program structure; this connection graph becomes the CFG. The CFG is the basis of many compiler optimizations and static-analysis tools.

The main function of the GFE module is to extract CFG information from the original PE file samples, keep the structure information, and further transform it into a Function Call Graph (FCG), as shown in the graph feature extraction part of Figure 4.1. The difference between CFG and FCG is that the CFG contains the assembly code of each basic block, while the FCG removes the assembly code of each basic block from the CFG. In the FCG, only information including the starting address of the basic block, Application Programming Interface (API) name, and function name are stored.

Static analysis is an important tool for malicious software detection. Specifically, the CFG structure information shows all the paths that a program will go through during the execution process. In addition, it graphically represents the possible flow of all basic blocks executed within a process and also reflects the basic blocks that are actually executed during execution. Some malicious software includes dead code (not executed from any execution path) as an obfuscation, but this process can alleviate these obfuscations.

Our feature extraction module is mainly to extract CFG structural information from the PE file. Because the CFG of a single sample contains a large amount of information, the pre-processing is very time consuming. Therefore, I only keep the structural relationship information of CFG, give each block a unified name, and use this name to represent this block. The assembly code in each block is not used as a feature. Each block in the CFG usually has two kinds of content, namely, the system API and the internal functions of the program. When a block is a system API, I directly use its name as the name of the block. If it is an internal function, I use the offset address of the function as the name of the block.

Graph Data Generation Module This module mainly uses the pre-trained language model to embed the text of a CFG block node. The FCG generated in the previous GFE module is a directed graph. Each node carries the corresponding text information and the relationship between each node has a direction. In order to train GNNs, it is necessary to convert the text information of each node into feature vectors of specific dimensions. Specifically, this module embeds the text names corresponding to CFG blocks through a large-scale pre-trained language model and generates graph structure data with node feature vectors.

Node Text Embedding Microsoft has released a pre-trained language model called MiniLM [51], which is based on the general method of reducing a large-scale transformer pre-trained model to a small model. This method is a kind of Deep Self-Attention Distillation (DSAD), which uses large-scale data for pre-training. The generation model used by our GDG module is called "all-MiniLM-L12-v2," which has a 1-billion-sized training set and is designed as a general model. The MiniLM model is a 12-layer transformer with a 384-hidden size and 12 attention heads that contains about 33 M parameters. It maps sentences and paragraphs to a 384-dimensional dense vector space and can be used for various tasks, such as clustering or semantic search. This model is the best in speed generation and performance among all pre-trained models that can generate 384-dimensional representation.

X_v is the feature vector of MiniLM generated from the CFG node text. Since the dimension of the generated feature vector is 384, the feature vector corresponding to the node of the CFG can be expressed as $X_v(384D)$.

Graph Data with Feature Vector In the previous step, I used the pre-trained model to generate a 384-dimensional dense vector for each node of the FCG. This vector is added to the corresponding nodes of the directed graph, and the complete graph data with the feature vectors of the nodes is generated.

I further initialize $h_v^0 = X_v$, where h_v^0 is the feature vector element of node v, and sever it as input data for the GC module. For example, the Main function as the first node of the graph, which corresponds to the text "Main," is converted into a 384-dimensional vector. It can be expressed as $h_1^0(384D)$.

Graph Classification Module This module mainly uses the GIN model to represent PE files as graph-level representations with fixed length. MLP detects malware by learning the graph-level representation of malicious and benign samples.

The GIN model focuses on constructing the graph-level representation. The node representation learned by the GIN can be directly used for tasks such as node classification and link prediction. For the graph classification task, the GIN can generate an embedding of the whole graph through the READOUT function. I designed a 3-layer structure of the GIN, where each layer is a GIN Convolutional (GINConv) layer. The GDG module generates $h_v^0(384D)$ as the input of GINConv1 and the output is $h_v^1(64D)$.

According to the preliminary GNN summary introduced in Section 4.2.1, I will introduce the specific operation in detail. The AGGREGATE function aggregates first-order neighborhood features, while the COMBINE function merges the features aggregated by neighbors with the current node features to update the current node features. The READOUT function converts all node features into graph-level features, mainly for graph classification. Common settings of the AGGREGATE function are sum, mean, and max. The sum function learns precise structural information, the mean is biased towards learning distribution information, and the max is biased towards learning representative element information. Because the injective condition of the mean and max functions is not met and it cannot distinguish graphs with some structures, the performance will be worse than that of sum; thus, I chose the sum setting. COMBINE is a direct sum or projective operator. If the AGGREGATE, COMBINE, and READOUT functions in the GNN are injective, the GNN can be as powerful as the Weisfeiler-Lehman test [52].

The GIN combines AGGREGATE (Eq.4.1) and COMBINE (Eq.4.2) into one step. I initialize $h_v^0 = X_v$ and $\mathcal{N}(v)$ is a set of nodes adjacent to v. I make ϵ as a learnable parameter or a fixed scalar. The initial value of ϵ, which is 0 by default. It is a hyperparameter we can tune, but probably not an essential one. Then, the GIN updates the node representations as

$$h_v^{(k)} = \mathrm{MLP}^{(k)}\left(\left(1+\epsilon^{(k)}\right)\cdot h_v^{(k-1)} + \sum_{u\in\mathcal{N}(v)} h_u^{(k-1)}\right). \tag{4.4}$$

To keep the feature dimensionality to a minimum, instead of using the GIN's original READOUT function by concatenating the results of all iterations/layers, I used only the output of the last layer. The final graph representations are the 3rd iteration/layer of the GIN and can be expressed as h_G. The READOUT function uses the summation.

Theoretically, with the increase in GINConv layers, the receptive field for each node embedding will gradually increase. Although our dataset is small, the 3-layer GINConv has obtained very good detection results. The specific structure, parameter information, and implementation details of the model will be discussed in the next section.

MLP Classifier The MLP has a 64-dimensional input and 2-dimensional output, with a total of three layers. The input layer has 64 dimensions, the hidden layer also has 64 dimensions, and the final output layer has 2 dimensions. In addition, the output of the hidden layer is batch normalized and the activation function is the Rectified Linear Unit (ReLU).

4.3 Implementation Details

The details of the implementation are introduced in this section. To verify the effectiveness of the proposed detection system, I used open-source libraries to implement it [1].

4.3.1 Malware Geometric Dataset

The malware datasets used in our previous studies were provided by other organizations, such as the EMBER and SOREL-20M datasets. These two datasets directly extract 2381-dimensional feature vectors from the PE files by surface analysis. Unfortunately, almost none of these datasets provide all the original PE files, including malicious and benign samples. Some datasets do not even provide the original PE files. Without benign samples, it is difficult to apply new feature extraction methods.

This prevents us from making improvements in the feature extraction stage and I can only use a fixed feature vector to improve different machine learning models. In addition, these datasets are not suitable for the GNNs because the previous feature extraction methods did not consider the structure information of PE files. Therefore, I made a dataset suitable for GNNs, which was named the Malware Geometric Binary Dataset (MGD-BINARY).

PE Files Source The benign PE file sample is obtained from the open-source project DikeDataset [2] and the malicious sample is obtained from the BODMAS malware dataset [3]. Some PE files were selected to build our dataset. The software types of all PE file samples used in our dataset are executable files under an x86-architecture Windows platform using a no Dynamic Link Library (DLL)-type.

Dataset Description The MGD-BINARY is mainly made for malware detection, i.e., malware binary classification. I collected a total of 1000 samples, including 500 malicious samples and 500 benign samples. From the BODMAS dataset, I selected 8 families of malware. In order to keep a balance between benign ware and malware, I selected a small sample from each of the 8 families to build the dataset and finally selected a total of 500 malware. The family distribution of malicious samples is shown in Table 4.1.

[1]Our code is available at: `https://github.com/kouunn/MalGIN`

[2]`https://github.com/iosifache/DikeDataset`

[3]`https://github.com/whyisyoung/BODMAS`

TABLE 4.1: Malware family distribution of MGD-BINARY.

Family Name	Category Name	Selected Count
sfone	worm	62
upatre	trojan	62
wabot	backdoor	62
benjamin	worm	62
musecador	trojan	63
padodor	backdoor	63
gandcrab	ransomware	63
dinwod	dropper	63
Total	-	500

I preprocessed the PE files through the GFE module and GDG module to get the MGD-BINARY. There are 1000 graph data and two kinds of manually labeled labels, among which the label of malicious samples is 1 and that of benign samples is 0. The dimensions of the feature vector of the graph node corresponding to each sample is 384. In addition, the average number of nodes and edges in the sample is 3861.75 and 5494.82, respectively. The statistical information is shown in Table 4.2.

TABLE 4.2: Graph statistics of MGD-BINARY.

Dataset	# Graphs	#Classes	#Features	Avg. #Nodes	Avg. #Edges
MGD-BINARY	1000	2	384	3861.75	5494.82

Dataset Splitting Due to the small scale of our dataset, if the ratio of the training set is too large, all evaluation metrics will approach 1 or equal 1, which will make the experimental results unable to be compared intuitively. For example, with 5-fold cross-validation, the ratios of the training set and the test set are 80% and 20%, respectively. Therefore, I used random permutations cross-validation (Shuffle & Split) to split our dataset and set the number of shuffling and splitting iterations to 5. The training set and testing set were set to 50% and 50%, respectively.

4.3.2 Graph Feature Extraction Module

Control Flow Graph The open-source angr[4] library can extract the CFG from a PE file. Angr is an open-source binary analysis platform for Python. It

[4]https://angr.io/

combines both static and dynamic symbolic ("concolic") analysis, providing tools to solve a variety of tasks.

Function Call Graph Module Because it is difficult to extract FCG directly, I can convert the extracted CFG into FCG. Each basic block in CFG contains a lot of assembly code. Instead of using the assembly code in each block, I extract the sub-function address and API call name corresponding to the CFG basic block. By using the extracted address and name, the original CFG is further converted into FCG.

4.3.3 Graph Data Generation Module

Pre-trained Language Model for Node Text Embedding SentenceTransformers [5] is a Python framework for state-of-the-art sentence, text, and image embeddings. The initial work is described in the Sentence-BERT (Sentence-Bidirectional Encoder Representations from Transformers) [53] paper. I used the MiniLM model provided by the SentenceTransformers library, which is called "all-MiniLM-L6-v2." The details of the pre-trained MiniLM model are shown in Table 4.3.

TABLE 4.3: Pre-trained MiniLM model details.

Name	all-MiniLM-L12-v2
Base Model	microsoft/MiniLM-L12-H384-uncased
Max Sequence Length	256
Dimensions	384
Normalized Embeddings	true
Size	120 MB
Pooling	Mean Pooling
Training Data	1B+ training pairs

Node Embedding Generation When I input text with a sequence length of less than 256 to the model, it will return a 384-dimensional sentence embedding vector. Then, the node embedding information generated by MiniLM is constructed into a directed graph with node attributes through the NetworkX [6] library.

[5] https://www.sbert.net/

[6] https://networkx.org/

4.3.4 Graph Classification Module

PyTorch Geometric (PyG) [7] is a library built upon PyTorch to easily write and train GNNs for a wide range of applications related to structured data. I used the PyG library version 2.0.2 to implement the malware detection experiment.

Our proposed GIN network has three layers and the output of each layer is 64 dimensions. The readout function sums the embedding of all nodes and generates the final representation of the graph. Each layer of the model is a GINConv layer, and the input and output dimensions of each GINConv layer are slightly different. The GINConv layer is composed of two fully connected layers. The output results of the first layer need to be normalized and the ReLU activation function applied before entering the second layer. GINConv uses the message passing mechanism to update the representation of each node. GINConv1 has an input dimension of 384 and a final output dimension of 64. GINConv2 and GINConv3 are identical, with both input and output dimensions of 64. The output dimension of the readout function is 64, so the final graph representation corresponding to each PE sample is a vector with 64 dimensions.

I used most of the default parameters of the GIN model [35]. Only the learning rate, batch size, and epochs were adjusted. The details of the GIN model training parameters are shown in Table 4.4. I trained the GIN model without excessive parameter tuning and set the learning rate at 0.0001 and the number of training epochs at 100. In the process of learning, because the default learning rate will cause the test set evaluation metrics curve to fluctuate greatly, I set a smaller learning rate to make the learning process smoother. To allow better generalization ability of the model, I set a larger size of 256 as the batch size. After our test, when the learning rate is set to be smaller and the number of epochs is improved, the result will improve slightly.

TABLE 4.4: GIN model training parameters.

Name	**Setting**
Loss Function	Negative Log-likelihood Loss
Activation Function	ReLU
Optimizer	Adam
Learning Rate	0.0001
Graph Pooling	Sum
MLP Dropout	0.5
Batch Size	128
Epochs	100

[7] https://www.pyg.org/

4.4 Experimental Evaluation

In this section, I apply the GIN model and discuss the experimental results.

4.4.1 Evaluation Metrics

In order to evaluate the performance of the proposed models, I used the following evaluation metrics. The detail of metrics such as TP and TN are described in Section 3.3.4

- **Accuracy** measures how often the model gets the prediction right. The formula for categorical accuracy is:

$$Accuracy = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}}.$$

- **Precision** is defined as:

$$Precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}}.$$

- **Recall** is defined as:

$$Recall = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}}.$$

- **F1-score** is defined as the harmonic mean of the precision and recall:

$$F1\text{-}score = 2 \times \frac{\mathrm{Precision} \times \mathrm{Recall}}{\mathrm{Precision} + \mathrm{Recall}}.$$

- **Area Under the ROC Curve (AUC)**, is one of the most important for measuring the performance of binary classification model. It is a performance measurement for a classification problem at various thresholds settings. The ROC Curve measures how accurately the model can distinguish between two things. AUC measures the entire two-dimensional area underneath the ROC curve. This score gives us a good idea of how well the classifier will perform.

which is the part under the ROC curve, is calculated as the calculus value of the ROC curve.

4.4.2 Evaluation Results

Our baseline method is LightGBM, which is an improved Gradient Boosting Decision Tree (GBDT). It is the same as the baseline algorithm of the EMBER [23] and SOREL-20M [24] datasets. The feature extraction method is version 2 of the EMBER dataset baseline algorithm. This method extracts 2381 features from PE samples and then uses LightGBM as a classifier. Both methods have been evaluated using MGD-BINARY. As shown in Table 4.5, I compared the results of our proposal with the baseline. In our evaluation, different feature extraction methods and classification methods are used for comparison in the same dataset (MGD-BINARY).

TABLE 4.5: Experimental results.

Feature Extraction Method		Classifier	Accuracy	F1-score	AUC
EMBER V2 (2381D)		LightGBM (Baseline)	0.99400 ± 0.00551	0.99388 ± 0.00546	0.99985 ± 0.00220
CFG + MiniLM +	GIN (3-layer, 32D)	LightGBM	0.97560 ± 0.01098	0.97650 ± 0.01054	0.97539 ± 0.01121
	GIN (3-layer, 64D)	LightGBM	**0.97840 ± 0.00662**	**0.97922 ± 0.00596**	**0.97808 ± 0.00683**
	GIN (3-layer, 128D)	LightGBM	0.97980 ± 0.00325	0.98030 ± 0.00306	0.97975 ± 0.00334
	GIN (3-layer, 256D)	LightGBM	**0.98040 ± 0.01148**	**0.98111 ± 0.01072**	**0.98021 ± 0.01171**
CFG + MiniLM +	GIN (3-layer, 32D)	MLP (2-layer, 32D)	0.96920 ± 0.00816	0.96971 ± 0.00813	0.96929 ± 0.00800
	GIN (3-layer, 64D)	MLP (2-layer, 64D)	**0.98640 ± 0.00496**	**0.98671 ± 0.00498**	**0.98630 ± 0.00501**
	GIN (3-layer, 128D)	MLP (2-layer, 128D)	0.98880 ± 0.00325	0.98908 ± 0.00311	0.98876 ± 0.00324
	GIN (3-layer, 256D)	MLP (2-layer, 256D)	**0.98920 ± 0.00204**	**0.98946 ± 0.00195**	**0.98919 ± 0.00217**

I evaluated the experimental results by Accuracy, F1-score, and AUC metrics. The Accuracy results show the baseline method is 0.99400 ± 0.00551 and our proposal under 3-layer of GIN and 64 dimensions of hidden layers is 0.98640 ± 0.00496. The F1-score metrics are 0.99388 ± 0.00546 and 0.98671 ± 0.00498, respectively. The AUC metrics are 0.99985 ± 0.00220 and 0.98630 ± 0.00501, respectively. As shown in Table 4.5, when the dimension is 256, I obtained the best result on the three metrics.

EMBER V2 + LightGBM is our baseline algorithm. From the results, as compared to the current state of the art, it has achieved very good results. First, the feature extraction method of EMBER V2 is very effective and, because the dimension is 2381, the high dimension better represents the features of the original information. As everyone knows, the LightGBM algorithm has excellent classification ability, which is also an important reason for its good results. Our proposed feature extraction method, due to the different dimensions of the final feature vectors, ranges from 32D to 256D and there are four cases. It can be seen that whether LightGBM or MLP is used as the classifier, the result will improve with the increase in feature vector dimensions. In addition, our proposed feature extraction method is based on GNNs. Under the same conditions, the results of LightGBM are not as good as those of MLP. Therefore, I conclude that LightGBM may be more effective for feature vectors based on statistics, rather than feature vectors with structural information extracted by GNNs. On the contrary, MLP is better at dealing with feature vectors with structural information extracted by GNNs. It should be noted that, when the feature vector dimension is low, for example, when the vector dimension is 32D, LightGBM is still superior to MLP.

I use the malware representation generated by the GIN model as a feature vector and use different types of machine learning algorithms to train the classifier. The evaluation results are shown in Table 4.6. Unexpectedly, the results of other classifiers are not as good as the MLP. The results of LightGBM are still stronger than most machine learning algorithms. The worst results were obtained from Random Forest. The Accuracy metrics for C-Support Vector Classification and Logistic Regression were similar. Logistic Regression was better in the F1-score, while C-Support Vector Classification was better in the AUC metric. The result changes largely between classification

TABLE 4.6: Comparison of experimental results under different classifiers.

Feature Extraction Method	Classifier	Accuracy	F1-score	AUC
EMBER V2 (2381D)	LightGBM (Baseline)	0.99400 ± 0.00551	0.99388 ± 0.00546	0.99985 ± 0.00220
CFG + MiniLM + GIN (3-layer, 256D)	Random Forest	0.93520 ± 0.001177	0.93865 ± 0.01094	0.93462 ± 0.01195
CFG + MiniLM + GIN (3-layer, 256D)	C-Support Vector Classification	0.97440 ± 0.01530	0.97475 ± 0.01533	0.97476 ± 0.01450
CFG + MiniLM + GIN (3-layer, 256D)	Logistic Regression	0.97440 ± 0.00320	0.97518 ± 0.00299	0.97417 ± 0.00352
CFG + MiniLM + GIN (3-layer, 256D)	LightGBM	0.98040 ± 0.01148	0.98111 ± 0.01072	0.98021 ± 0.01171
CFG + MiniLM + GIN (3-layer, 256D)	MLP (2-layer, 256D)	**0.98920 ± 0.00204**	**0.98946 ± 0.00195**	**0.98919 ± 0.00217**

methods, so I estimated that the classification method tuned for prior characteristics is not suitable for GIN-based feature extraction. I estimated that there is a large tuning space on both the GIN side and the classification side, so I hope that many researchers challenge the research in this area.

TABLE 4.7: Additional experimental results.

Feature Extraction Method		Classifier	Accuracy	F1-score	AUC
CFG + MiniLM +	GIN (4-layers, 32D)	MLP (2-layer, 32D)	0.97200 ± 0.00912	0.97304 ± 0.00851	0.97182 ± 0.00932
	GIN (4-layers, 64D)	MLP (2-layer, 64D)	0.98880 ± 0.00449	0.98914 ± 0.00430	0.98871 ± 0.00451
	GIN (4-layers, 128D)	MLP (2-layer, 128D)	0.98920 ± 0.00240	0.98945 ± 0.00244	0.98918 ± 0.00253
	GIN (4-layers, 256D)	MLP (2-layer, 256D)	0.99080 ± 0.00204	0.99104 ± 0.00190	0.99072 ± 0.00211
CFG + MiniLM +	GIN (5-layers, 32D)	MLP (2-layer, 32D)	0.96920 ± 0.01197	0.96954 ± 0.01240	0.96953 ± 0.01127
	GIN (5-layers, 64D)	MLP (2-layer, 64D)	0.98560 ± 0.00196	0.98598 ± 0.00183	0.98551 ± 0.00197
	GIN (5-layers, 128D)	MLP (2-layer, 128D)	0.98920 ± 0.00483	0.98953 ± 0.00455	0.98911 ± 0.00497
	GIN (5-layers, 256D)	MLP (2-layer, 256D)	**0.99160 ± 0.00427**	**0.99189 ± 0.00396**	**0.99148 ± 0.00455**

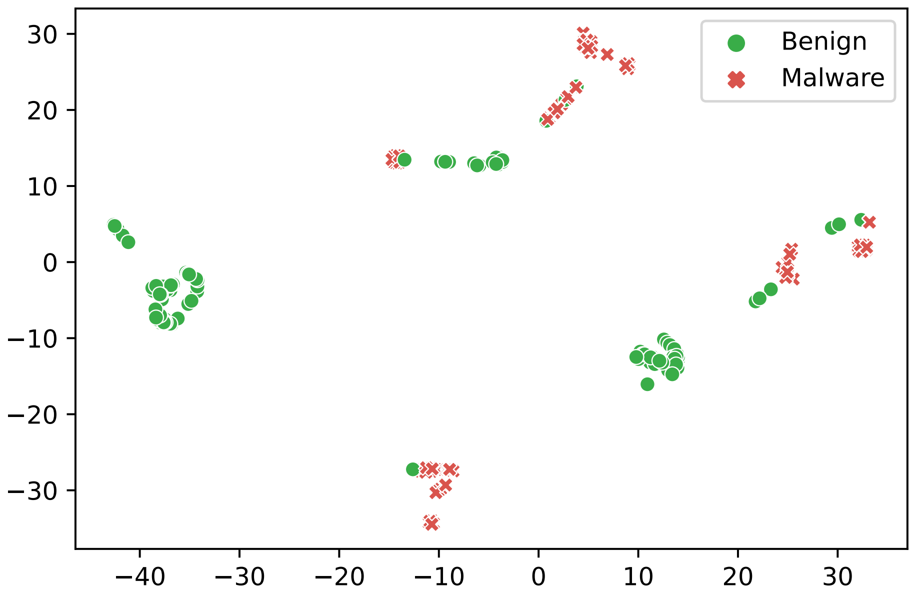
In addition, I verified the performance of the GIN model classification results under different layer numbers and hidden layer dimensions of the GINConv layer. The evaluation results are shown in Table 4.7. With the increase of the GIN model layers and hidden layer dimensions, I found that the results improve. This proves the effectiveness of extracting sample features based on CFG and using GNNs to detect malware.

4.5 Discussion

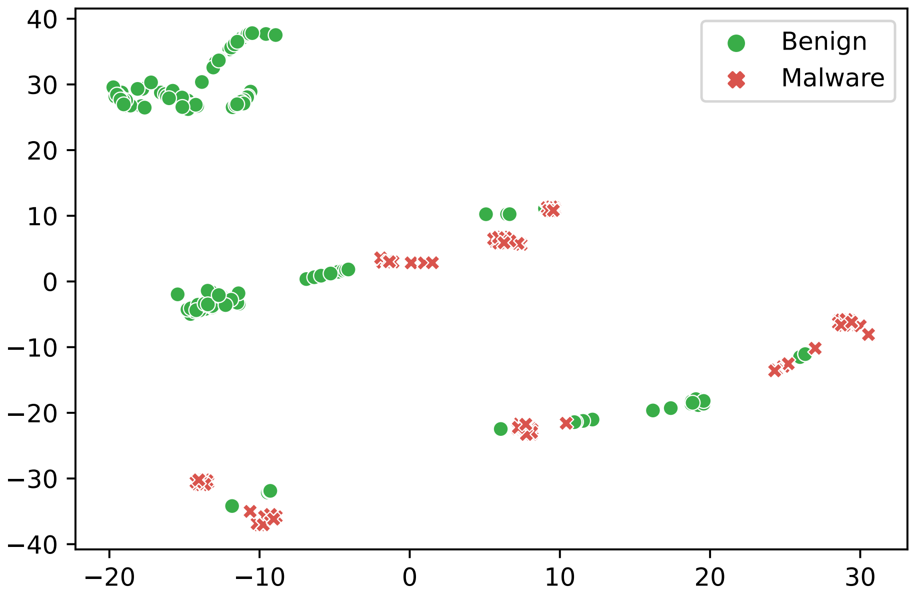
In this section, I discuss the limitations of our method and show the advantages of our proposal by t-Distributed Stochastic Neighbor Embedding (t-SNE) visualization technology. In addition, I summarize the main factors of our approach to improve performance.

I currently face the following primary limitations. First, MGD-BINARY has fewer samples; thus, the model cannot learn more samples. Benign samples involve copyright issues. Most public malware datasets do not directly provide the original PE files of benign samples, so it is difficult for us to collect large-scale benign samples. Second, the process of extracting CFG from samples is very time consuming, which greatly increases the time cost of building large-scale graph datasets.

I visualized the feature vectors of the baseline and our proposal using t-SNE technology. As shown in Figure 4.2 and Figure 4.3 , I visualized the training set and test set of the two methods, respectively. Whether it is the



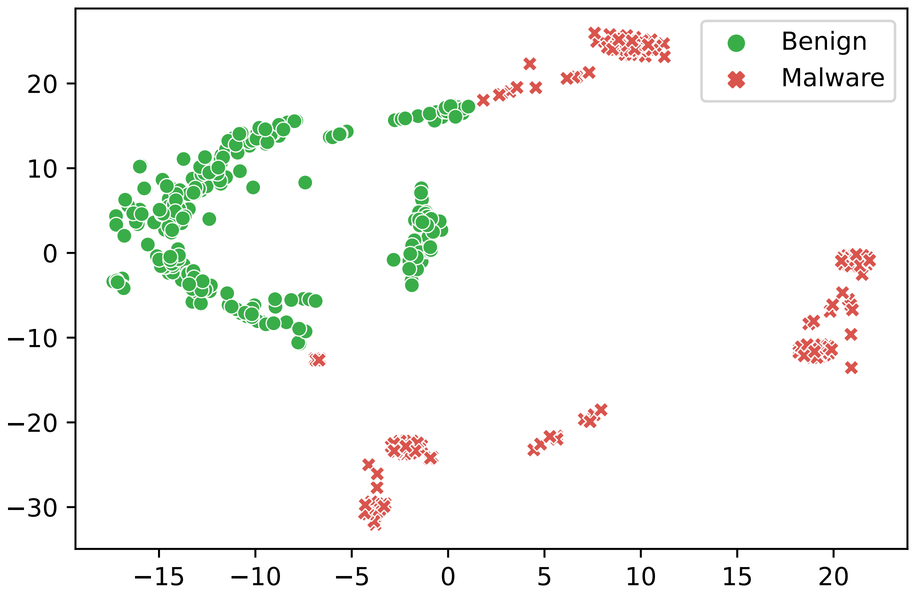
(A) EMBER V2 (2381D) Training Set



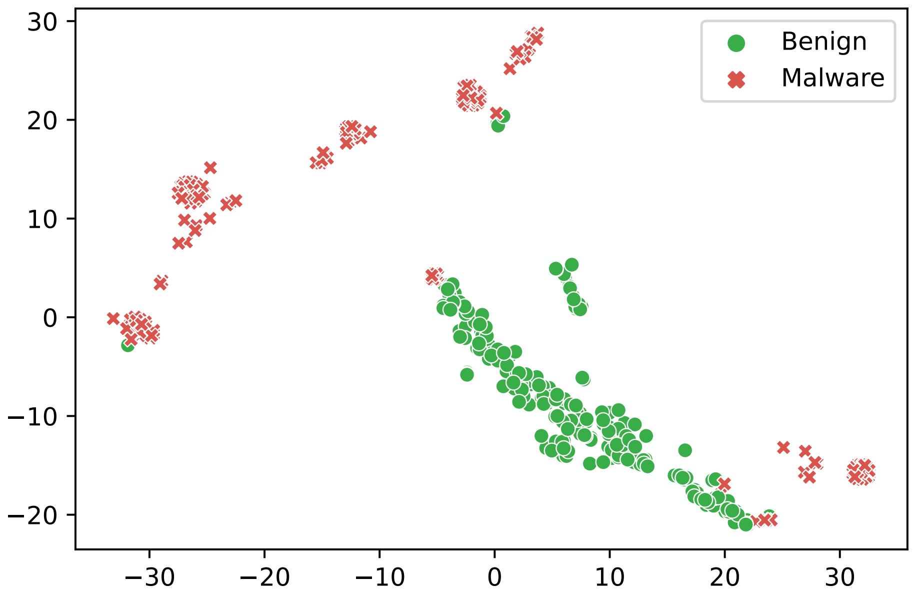
(B) EMBER V2 (2381D) Test Set

FIGURE 4.2: t-SNE visualization of Baseline.

training set or the test set, the benign samples in our proposal are more compactly clustered and distant from the malware, while the benign samples and malware in the baseline are more dispersed. The more compact distribution of benign samples allows the model to learn better; when there are additional



(A) CFG + MiniLM+ GIN (64D) Training Set



(B) CFG + MiniLM+ GIN (64D) Test Set

FIGURE 4.3: t-SNE visualization of Proposal.

malicious samples in the training set, the model can better distinguish and learn, thus improving the effectiveness of the malware detection.

Regarding the main factors for improving the detection performance, I find the following three points: First, extracting CFG from a PE file can effectively extract the function call relationship and program structure information. Second, by using a MiniLM pre-trained language model, the node information of the CFG can be effectively represented by fixed-length feature vectors. Finally, due to the superiority of the GNN itself, each node can fully learn the information of its neighbors through the message passing mechanism and finally obtain low dimensional graph-level representation, making it easier for the downstream detection and classification tasks to achieve good results.

4.6 Conclusion

This chapter proposed a new method of extracting malware features based on the GNN and investigated the performance of malware detection under different classification models. I extracted the CFG from PE files and then completed the construction of a graph dataset through GFE, GDG, and GC modules. Finally, I used the MLP model to detect malware. In addition, I also investigated the effects of other classification models, such as LightGBM and Random Forest. Based on our experimental results, MLP had the best performance. CFG is a directed graph. The advantages of graph data are that each node has a different feature and it contains the call structure information of PE samples. Through the GFE and GDG modules, I created a dataset suitable for GNNs and named it MGD-BINARY. By adjusting the GIN model in the GC module, I can generate malware representations with different dimensions. Compared with the traditional feature extraction methods, the dimensions of our feature representation are compressed lower. The advantage of the GIN model lies in compressing the high-dimensional features of each node in the graph into the low-dimensional space and constructing the representation of the whole graph with low dimensions for each graph sample. The GIN-based malware detection model has great potential with the increase of layers and hidden dimensions. In future work, I plan to further expand the dataset, especially by adding more benign samples. I will also seek to raise the graph extraction speed of PE files and further improve the evaluation results of the model.

Chapter 5

Graph Contrastive Learning for Malware Classification

5.1 Introduction

Fueled by the progress of software technology and the internet's development, thousands of malware are created every day due to the proliferation of malware creation and obfuscation tools. Such a massive flood of data poses a considerable challenge to malware analysts and Security Operation Centers (SOCs). Traditional malware detection methods cannot continue to quickly and effectively detect such a massive amount of newly created malware. In past decades, machine learning has played an important role in information security, especially in malware detection and classification tasks. It is also a promising method to detect and classify large-scale newly created malware using the features of samples.

In the field of static malware detection, the feature extraction method of Portable Executable (PE) files used in the Endgame Malware Benchmark for Research (EMBER) dataset [23] has been widely applied. This feature extraction method directly provides consistent feature vectors to researchers, allowing individuals in the same field to compare their respective proposed methods. The information related to software structure, such as the Control-Flow Graph (CFG), is rarely extracted, and most methods are based on surface analysis for extracting statistical information as features. In addition, in most malware detection and classification scenarios, the model is supervised for end-to-end training.

Supervised learning requires manual labeling of a large amount of data, and the model effect depends on the quality of the labels. Therefore, the future research trend, which is exploring unsupervised learning methods, is critical for malware detection and classification. In recent years, Graph Neural Networks (GNN) have made remarkable progress. I can exploit their powerful representation ability to better represent malware and improve the effectiveness of its detection and classification. However, one remaining difficulty is how to represent malware in graphical form. Since CFG is a natural graph structure, I can generate the graph structure data of malware by extracting CFG. Therefore, I seek to classify malware by constructing a graph dataset and using unsupervised learning. Since no publicly available graph classification dataset exists for malware classification, I started by creating such a dataset.

Our contributions can be summarized as follows:

- I propose a malware classification framework based on graph contrastive learning under unsupervised learning.
- I retain the structural information of the samples extracted from CFG and embed the text features of each node with a pre-trained language model.
- I create a special graph dataset for malware classification that can be used directly on GNNs.
- Our pre-trained model can effectively perform a low-dimensional representation of malware with which a variety of downstream tasks can be performed. I have achieved good results on malware family classification tasks.

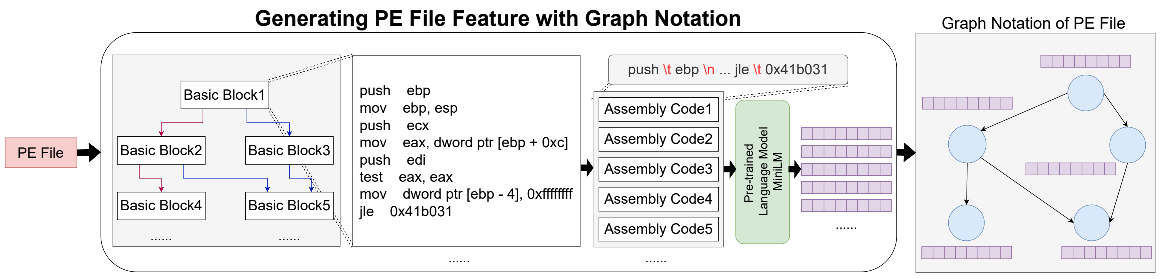
The remainder of this chapter is organized as follows. Section 5.2 introduces the principles of our proposed data augmented GraphCL-based static malware PE classification system and its application to malware classification. In Section 5.3, I briefly discuss the implementation details of our proposal. In Section 5.4, I describe the corresponding experiments and evaluate their feasibility as well as the advantages of our proposal. In Section 5.5, I discuss the current limitations of our proposal. Finally, I discuss our conclusion and describe future work in Section 5.6.

5.2 Proposed Malware GraphCL with Data Augmentations

Our proposal is a data augmented GraphCL-based static malware PE classification framework, which can obtain a graph-level representation from malware. I directly extract malware CFG from PE files and through graph contrastive learning obtain a representation of the malware with a vector notation. Finally, malware representations can be performed downstream for various tasks. Graph-level representation shows good performance on malware classification tasks. Next, I scrutinize the framework.

5.2.1 Raw Graph Generation

To train the GNNs, I need to produce graph datasets, and the main task of this module is to convert PE files into raw graphs. The overview of raw graph generation is shown in Figure 5.1. This raw graph generation is basically the same as that of Figure 4.1 introduced in Section 4.2.2. The difference is that the current method additionally uses assembly codes in basic blocks as the features of graph nodes.



```
push    ebp
mov     ebp, esp
push    ecx
mov     eax, dword ptr [ebp + 0xc]
push    edi
test    eax, eax
mov     dword ptr [ebp - 4], 0xffffffff
jle     0x41b031
```

FIGURE 5.1: Raw graph generation for proposal

CFG Structure and Disassembly Code First, the CFG information is extracted from the original PE file samples, the structure information of the basic blocks is retained, and the disassembly code of each basic block is extracted. Each basic block of CFG has a corresponding disassembly code, and the relationship between each basic block is directional. Disassembly codes need to be transformed into feature vectors of specific dimensions to train GNNs. Since the malware CFG is usually a very large graph, extracting the CFG is very time consuming. Since the disassembly code in each basic block of CFG contains rich semantic information, I need to completely exploit that information and suitably embed it, for example, using a large pre-trained language model.

Pre-trained Language Model MiniLM MiniLM is a method released by Microsoft, which based on reducing large-scale transformer pre-trained models into smaller models [51]. This Deep Self-Attention Distillation (DSAD) method uses large-scale data for pre-training. The model I use is called "all-MiniLM-L12-v2," which has a 1-billion-sized training set and is designed as a general-purpose model. MiniLM model is a 12-layer transformer with a 384 hidden size and 12 attention heads that contain about 33 M parameters. It maps sentences and paragraphs to a 384-dimensional dense vector space and can be used for tasks like clustering or semantic search. This model is the fastest generation of related studies and still provides good quality. In this step, a 384-dimensional dense vector is generated for each CFG node using the pre-trained model. This vector is added to the corresponding nodes of the directed graph to generate complete graph data with node feature vectors. These directed graphs are used as our raw graph data.

5.2.2 Data Augmentation for Graphs

I used the following four data augmentation methods. As shown in Figure 5.2, our proposal uses two of them. The best combination is explored in Section 5.

Node Dropping Randomly discard some parts of the vertex and its connections. The missing parts of the vertices do not affect the semantic meaning

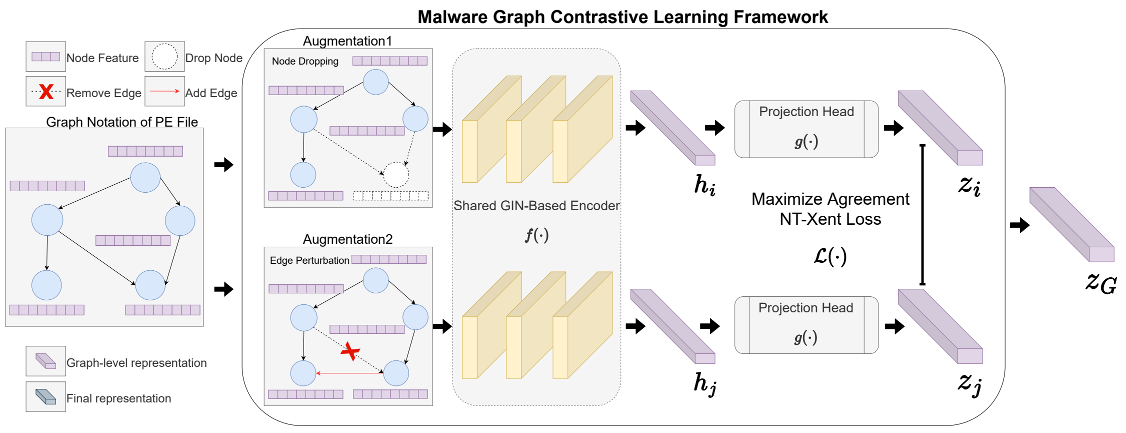


FIGURE 5.2: Proposed malware graph contrastive learning framework for graph representation generation

of the graph, and so the learned representation is consistent under the disturbance of nodes. The dropping probability of each node follows a default Bernoulli distribution (or any other distribution).

Edge Perturbation Randomly add or remove a certain ratio of edges so that the learned representation is consistent under edge perturbation. The prior information of the representation is that adding or removing some edges does not affect the semantics of the graph. The adding or removing probability of each edge also follows a default Bernoulli distribution. I only used Edge Removing in this evaluation.

Attribute Masking The attribute information of some nodes is randomly removed, which urges the model to use other information to reconstruct the masked node attributes. The masking probability of each node feature dimension follows a default uniform distribution. I only used simple Feature Masking.

Subgraph Sampling Use random walk subgraph sampling [54] to extract subgraphs from the original graph. The basic assumption is that a graph's semantic information can be preserved in its local structure.

Table 5.1 overviews the data augmentation for graphs. The default augmentation (dropping, perturbation, masking) ratio is set to 0.1, and the walk length is set to 10.

TABLE 5.1: Overview of data augmentation for graphs

Data Augmentation	Type	Default Setting
Node Dropping	Nodes, edges	Bernoulli distribution (ratio = 0.1)
Edge Perturbation	Edges	Bernoulli distribution (ratio = 0.1)
Attribute Masking	Nodes	Uniform distribution (ratio = 0.1)
Subgraph Sampling	Nodes, edges	Random Walk (length = 10)

5.2.3 Graph Contrastive Learning

Motivated by recent developments in graph contrastive learning, I propose a graph contrastive learning framework for malware classification. As shown in Figure 5.2, in graph contrastive learning, pre-training is performed by maximizing the agreement between two augmented views of the same graph by contrastive loss in the potential space. The framework consists of the following four main components:

Graph Data Augmentation Throughout the GraphCL framework, given graph data G, two related augmented graphs, $\hat{G}_i, \hat{G}_j$, are generated as positive sample pairs by data augmentation.

GIN-based Encoder GIN-based encoder $f(\cdot)$ is used to generate graph-level vector representation. There are three layers in the GIN-based encoder, and the hidden layer has 64 dimensions. Through the readout function, the embedding of all the nodes is summed to obtain initial graph representation h_i, h_j for augmented graphs $\hat{G}_i, \hat{G}_j$. Graph contrastive learning does not apply any constraint to the GIN-based encoder.

Projection Head Nonlinear transformation $g(\cdot)$, called a projection head, maps the augmented representations to another latent space. Contrastive loss is computed in the latent space, and z_i, z_j are obtained by applying a two-layer perceptron (MLP).

Contrastive Loss Function Contrastive loss function $\mathcal{L}(\cdot)$ is defined to maximize agreement between positive pairs z_i, z_j and negative pairs. As illustrated in Figure 5.3, A simple framework for contrastive learning of visual representations. Two separate data augmentation operators are sampled from the same family of augmentations ($t \sim T$ and $t' \sim T$) and applied to each data example to obtain two correlated views. Here I exploit the normalized temperature-scale cross-entropy loss (NT-Xent) [55][56].

After training is completed, I throw away the projection head $g(\cdot)$ and use encoder $f(\cdot)$ and obtain a graph-level final representation of z_G for downstream tasks.

5.2.4 Graph Classification

By pre-training with GraphCL, I can obtain a valid graph representation z_G. To further verify the effectiveness of our method, different classification models can be chosen for the process, such as random forest, logistic regression, SVM, and so on. I chose C-Support Vector Classification (SVC) as the algorithm to validate our pre-trained model's effectiveness.

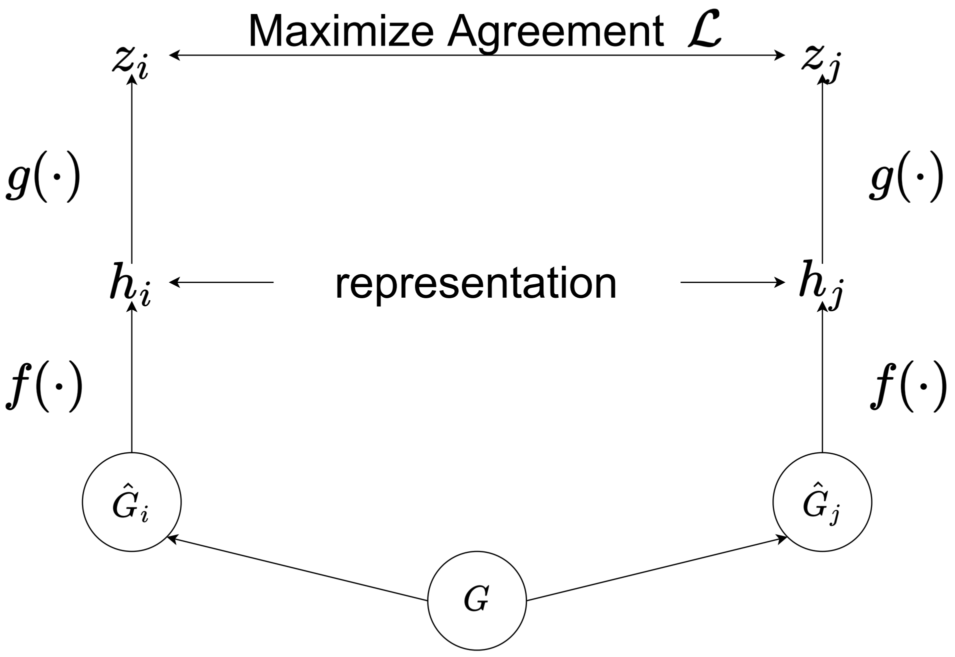


FIGURE 5.3: A simple framework for contrastive learning of visual representations.

5.3 Implementation Details

I verified the effectiveness of our proposed contrastive learning framework by implementing it with open-source libraries. The implementation details are introduced in this section.

5.3.1 Malware Geometric Multi-Class Dataset

PE Files Source The PE file sample was obtained from the BODMAS Malware Dataset [25]. The software types of all the PE file samples used in our dataset are executable files under an x86-architecture Windows platform without any Dynamic Link Library (DLL) type.

Dataset Description From the BODMAS dataset, I selected eight families of malware and took 500 samples from each family, for a total of 4000 samples in our dataset. Our dataset is named MGD-MULTI. The malware family distribution information is shown in Table 5.2.

Due to the difficulty of collecting benign samples and the imbalanced data problem, I did not include white samples in our multi-class dataset. In Section 4.3, the MGD-BINARY has been introduced. It contained benign samples. I used almost the same GIN model to represent the PE samples, with a slightly different operation of the READOUT layer this time compared to the GIN model in our previous work, giving the final representation a higher vector dimensionality. Based on our previous research [49], I believe that the GIN model can effectively distinguish benign samples from malicious ones. In future work, I will add benign samples to our dataset.

TABLE 5.2: Malware family distribution of MGD-MULTI

Family Name	Category Name	Origin Count	Selected Count	Graph Data Size
sfone	worm	4729	500	3.2 GB
upatre	trojan	3901	500	879.4MB
wabot	backdoor	3673	500	4.1 GB
benjamin	worm	1071	500	263.1MB
musecador	trojan	1054	500	1.5 GB
padodor	backdoor	655	500	2.9 GB
gandcrab	ransomware	617	500	6.6 GB
dinwod	dropper	509	500	3.3 GB
Total	-	16209	4000	22.7 GB

Among the different types of malware, I chose families that are more common and have a relatively large number in BODMAS. Due to some limitations of the CFG extraction tool for the PE files I used, many samples couldn't be recognized, causing extraction failure. In addition, for large PE file samples, the process of extracting CFG is very time-consuming. Since the extraction of some samples will fail, I selected a family with more than 500 samples in BODMAS and relatively small original PE files. I further improved the efficiency by only selecting successful samples whose total extraction time is less than 20 seconds in which the total extraction time includes the time of the feature vectors generated by the pre-trained language model. I finally got our MGD-MULTI whose extracted graph data statistical information is shown in Table 5.3.

TABLE 5.3: Graph statistics of MGD-MULTI

Dataset	# Graphs	#Classes	#Features	Avg. #Nodes	Avg. #Edges
MGD-MULTI	4000	8	384	3861.75	5494.82

Dataset Splitting I split 4000 pieces of data in MGD-MULTI into training, validation, and testing sets of 50%, 20%, and 30%, respectively. Since the results of the validation set and the test are similar, only the test set results are shown.

5.3.2 Pre-trained Language Model MiniLM

SentenceTransformers is a python framework for state-of-the-art sentence, text, and image embeddings. The initial work was described in a paper from the Sentence-Bidirectional Encoder Representations from Transformers (Sentence-BERT) [57]. I used the MiniLM model provided by the Sentence-Transformers library with the model name, all-MiniLM-L6-v2. The model details are shown in Table 5.4.

TABLE 5.4: Pre-trained MiniLM model details

Name	**all-MiniLM-L12-v2**
Base Model	microsoft/MiniLM-L12-H384-uncased
Max Sequence Length	256
Dimensions	384
Normalized Embeddings	true
Size	120 MB
Pooling	Mean Pooling
Training Data	1B+ training pairs

GraphCL Model PyGCL [58] is a PyTorch-based open-source Graph Contrastive Learning (GCL) library, which features modularized GCL components from published papers, standardized evaluation, and experiment management. The batch_size of all the experiments is 128, and the optimizer is Adam with a learning rate of 0.0001.

5.4 Evaluation

In this section, I apply the GraphCL model and discuss the experiment results and limitations of our method.

5.4.1 Evaluation Metric

I used the following evaluation metrics to assess the performance of our proposed models:

- **The Micro-averaged F1 score** is defined as the harmonic mean of the precision and recall:

$$MicroF1\text{-}score = 2 \times \frac{\text{Micro-Precision} \times \text{Micro-Recall}}{\text{Micro-Precision} + \text{Micro-Recall}}$$

  Micro-Precision is the sum of all true positives divided by the sum of all true positives and false positives. Micro-Recall is the sum of true positives for all classes divided by actual positives.

- **The Macro-averaged F1 score** is defined as the mean of the class-wise/label-wise F1-scores:

$$MacroF1\text{-}score = \frac{1}{N} \sum_{N}^{i=0} F1\text{-}score_i$$

  where i is the class/label index and N is the number of classes/labels. F1-score is defined as the harmonic mean of the precision and recall.

5.4.2 Evaluation Results

Next, I apply the GraphCL model and discuss the experiment results of our method.

Different Data Augmentation Combination Results I selected five different data augmentation methods: Identical (I), Edge Removing (ER), Node Dropping (ND), Feature Masking (FM), and Random Walk Subgraph (RWS). To compare the different data augmentation approaches on the GraphCL model, I used both data augmentation approaches for the input graph itself (Identical + Identical) as the GraphCL model baseline. I also tried different combinations of data augmentation, such as ER and ND, FM and ND, FM and ER, RWS and ER, RWS and ND, and RWS and FM. The experimental results are shown in Table 5.5. The best two data augmentation combinations were RWS and FM. I obtained the best Micro-F1 (0.9958) and Macro-F1 (0.9959).

TABLE 5.5: different augmentation combinations

Method (+SVC)	Augmentation[1]	Micro-F1	Macro-F1
GraphCL	I + I	0.9883	0.9883
GraphCL	ER + ND	0.9925	0.9924
GraphCL	FM + ND	0.9942	0.9942
GraphCL	FM + ER	0.9942	0.9942
GraphCL	RWS[2]+ ER	0.9950	0.9949
GraphCL	RWS + ND	0.9950	0.9949
GraphCL	RWS + FM	**0.9958**	**0.9959**

[1] Default ratio setting is 0.1.
[2] RWS uses a default walk length setting of 10.

Best Combination with Different Ratio Results In the previous set of experiments, I found that the best data augmentation combination is RWS + FM. Based on this combination, I also investigated the results on different ratios on the FM side, and the FM results on different ratios are shown in Table 5.6.

TABLE 5.6: Best combination with different ratio results

Method (+SVC)	Augmentation (Ratio)	Micro F1	Macro F1
GraphCL	RWS[1]+ FM (0.1)	0.9958	0.9959
GraphCL	RWS + FM (0.2)	0.9967	0.9967
GraphCL	RWS + FM (0.3)	**0.9975**	**0.9976**
GraphCL	RWS + FM (0.4)	0.9958	0.9958
GraphCL	RWS + FM (0.5)	0.9942	0.9941

[1] RWS uses a default walk length setting of 10.

Comparison of Different Methods All of our previous studies focused on supervised learning. This study is a graph contrastive learning method in an unsupervised setting.

Baseline 1 is a direct graph-level encoding of an input graph using GIN as an encoder, and then the embedding effect is evaluated using SVC. Baseline 2 is data augmentation using the input graph itself. Baseline 3 is our proposal described in Chapter 4. Train a GIN model under supervised setting, with a two-layer MLP directly connected after the readout layer.

A comparison of different methods is shown in Table 5.7. GraphCL with a setting of RWS + FM (0.3) achieved the best classification results.

TABLE 5.7: Comparison of different methods

Name	Method	Type	Micro-F1	Macro-F1
Baseline 1	GIN-Encoder + SVC	U[1]	0.9617	0.9620
Baseline 2	GraphCL (I + I) + SVC	U	0.9883	0.9883
Baseline 3	GIN + MLP (Previous work [49])	S[2]	0.9958	0.9957
Proposal	GraphCL (RWS + FM_0.3) + SVC	U	**0.9975**	**0.9976**

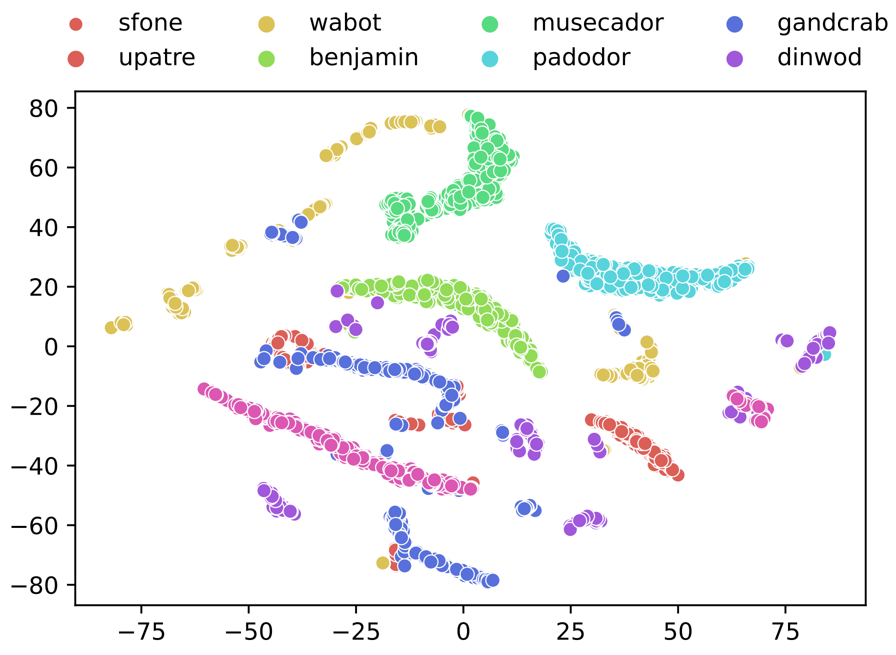
[1] U denotes unsupervised learning.
[2] S denotes supervised learning.

I used t-SNE technology to visualize the embedding of Baseline 1 and our proposed method. As shown in Figure 5.4, the method of Baseline 1 has already clustered some categories, such as the malware of the "padodor" family, but it cannot cluster the "gandcrab" family well. On the other hand, our comparative learning model proposal can better cluster different categories in the eight classes, and a large distance between different categories is maintained.

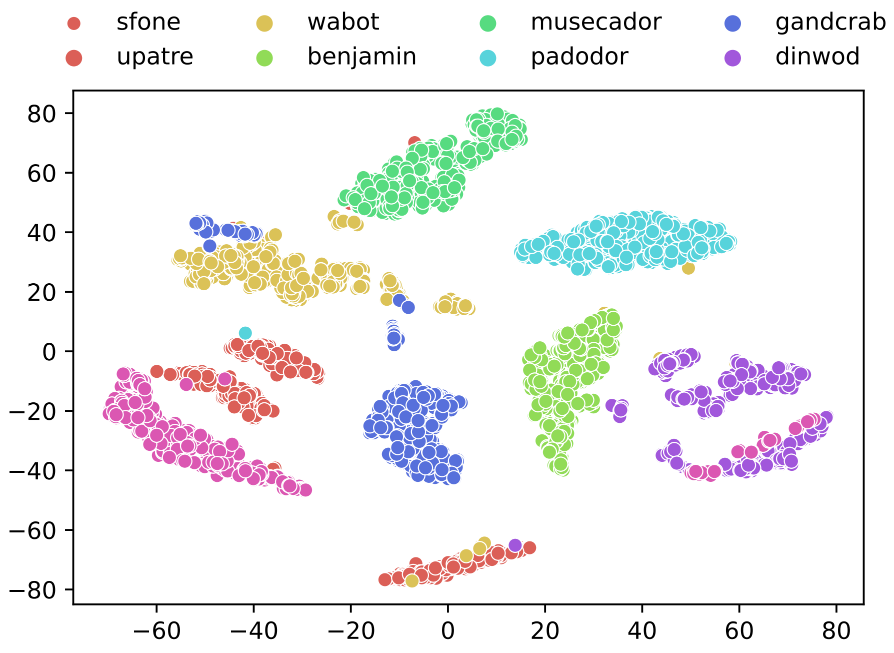
5.5 Discussion

GraphCL (I + I) is a combination of two Identical, and the effect is equivalent to turning a training set of N samples into 2N samples. The same data model is learned twice for the same data, so the obtained result naturally outperforms GIN-Encoder. The RWS + FM method is most effective because neither method changes the structural information of the original graph. The RWS method samples a subgraph that is smaller than the structure of the original graph, but still retains most of the original graph's structure. For the FM method, the original graph structure is not changed at all, but the values of some dimensions of the node feature vectors are masked, which makes the node features more robust. On the contrary, the other two methods (ER and ND) change the original graph structure more, so the results are lowered.

Because of the relatively large graph structure I extracted from the PE file and the high dimensionality of the nodes in each graph (384 dimensions), our result still leads to a slow training of the GraphCL model even though the dataset size is not too large, only 4000 pieces of data.



(A) Baseline 1:
GIN-Encoder + SVC



(B) Proposal:
GraphCL (RWS+FM_0.3)

FIGURE 5.4: t-SNE visualization of Baseline 1 and Proposal

The training stage requires around ten minutes with NVIDIA GeForce RTX 3090. If we use more memory and faster GPU devices, we can further increase the batch size, and the model can get better generalization. At the same time, the training time can be further shortened and the scale of data set can be further expanded. I desire a better way to generate node features, such as a lower dimensional in a method that retains its effectiveness.

5.6 Conclusion

I proposed the unsupervised learning of different families of malware using graph contrastive learning and the multi-classification of learned vectors using SVC and obtained good results. I extracted the CFG of the malware, embedded the disassembly code in a basic block through a large pre-trained language model MiniLM, and obtained a directed graph with node features. The advantage of a directed graph is that it contains the call structure information of the sample in addition to the features of each node. I also produced a multi-classification dataset: MDG-MULTI. Unsupervised GraphCL-based malware classification methods have surpassed graph-based supervised learning methods, such as the Graph Isomorphism Network (GIN) for graph classification. In future work, I will shift our focus to unsupervised learning.

Chapter 6

Conclusion

This thesis set out to apply AI technology for malware detection and classification. Improve the main problems in this field, such as robustness, interpretability and concept drift of malware machine learning models. How to reduce false alarms is also a special problem. In order to solve these problems, I mainly made detailed investigations in three aspects.

Reduce False Alarms This study proposed a custom log loss function with α and β parameters for the LightGBM algorithm to solve the malware detection problem. I extracted 27 valid features from a non-public FFRI dataset for malware detection. I validated the effectiveness of our proposed method by separately and simultaneously evaluating it on FFRI and another public dataset, EMBER. Our result shows that the custom log loss function can reduce FP more than the normal log loss function. However, since the custom log loss function increased FN more than the normal log loss function, I also proposed a hybrid usage of the Custom_AUC and Custom_FPR models to prioritize the positive results. Reducing the FPR by the custom log loss function will significantly lower the priority given to false alarms, thus alleviating the pressure on security responders. Although I can control the model to learn more difficult samples by customizing the loss function, learning them is inherently difficult for the model. It is difficult to achieve complete learning; it inevitably leads to a certain number of FPs and FNs. Our proposed method penalizes FP and FN in the training phase and can only make them as low as possible instead of zero. I believe that the fundamental reason lies in the data themselves, the size of the dataset, and the effectiveness of the extracted features.

Interpretability This study proposed a new method of extracting malware features based on the GNN and investigated the performance of malware detection under different classification models. I extracted the CFG from PE files and then completed the construction of a graph dataset through GFE, GDG, and GC modules. Finally, I used the MLP model to detect malware. In addition, I also investigated the effects of other classification models, such as LightGBM and Random Forest. Based on our experimental results, MLP had the best performance. CFG is a directed graph. The advantages of graph data are that each node has a different feature and it contains the call structure information of PE samples. Through the GFE and GDG modules, I created a dataset suitable for GNNs and named it MGD-BINARY. By adjusting the GIN model in the GC module, I can generate malware representations with different dimensions. Compared with the traditional feature extraction methods, the dimensions of our feature representation are compressed lower. The advantage of the GIN model lies in compressing the high-dimensional features of each node in the graph into the low-dimensional space and constructing the representation of the whole graph with low dimensions for each graph sample. The GIN-based malware detection model has great potential with the increase of layers and hidden dimensions. In future work, I plan to further expand the dataset, especially by adding more benign samples. I will also seek to raise the graph extraction speed of PE files and further improve the evaluation results of the model.

Self-supervised Graph Contrastive Learning This study proposed the self-supervised learning of different families of malware using graph contrastive learning and the multi-classification of learned vectors using SVC and obtained good results. I extracted the CFG of the malware, embedded the disassembly code in a basic block through a large pre-trained language model MiniLM, and obtained a directed graph with node features. The advantage of a directed graph is that it contains the call structure information of the sample in addition to the features of each node. I also produced a multi-classification dataset: MDG-MULTI. Unsupervised GraphCL-based malware classification methods have surpassed graph-based supervised learning methods, such as the Graph Isomorphism Network for graph classification. In future work, I will shift our focus to unsupervised learning.

The first research has identified custom loss function can significantly reduce the false alarm rate of the model while maintaining a high detection rate. I can artificially control the balance between the detection rate and false alarm rate of the model. The second major finding was that compared with the traditional statistical features of malware, graph representation learning based on CFG features can better learn the structural features of malware, and relying on powerful graph neural network, excellent detection results can be determined. One of the more significant findings to emerge from this study is that compared with the supervised multi-class classification model, self-supervised graph contrastive learning can achieve better results with a small number of samples and no need to label the family information of malware training data.

Bibliography

[1] A. Test, *Malware Statistics & Trends Report | AV-TEST*, 2022. [Online]. Available: https://www.av-test.org/en/news/facts-analyses-on-the-threat-scenario-the-av-test-security-report-2019-2020/.

[2] J. O. Kephart, G. B. Sorkin, W. C. Arnold, D. M. Chess, G. Tesauro, and S. R. White, "Biologically Inspired Defenses Against Computer Viruses," in *Proc. 4th International Joint Conference on Artificial Intelligence(IJCAI 1995)*, pp. 985–996, 1995. [Online]. Available: https://www.ijcai.org/Proceedings/95-1/Papers/127.pdf.

[3] G. McGraw and J. G. Morrisett, "Attacking malicious code: A report to the infosec research council," *IEEE Softw.*, vol. 17, no. 5, pp. 33–41, 2000. DOI: 10.1109/52.877857.

[4] M. Christodorescu, S. Jha, J. Kinder, S. Katzenbeisser, and H. Veith, "Software transformations to improve malware detection," *J. Comput. Virol.*, vol. 3, no. 4, pp. 253–265, 2007. DOI: 10.1007/s11416-007-0059-8.

[5] E. Raff, R. Zak, R. Cox, *et al.*, "An investigation of byte n-gram features for malware classification," *J. Comput. Virol. Hacking Tech.*, vol. 14, no. 1, pp. 1–20, 2018. DOI: 10.1007/s11416-016-0283-1.

[6] Y. Nagano and R. Uda, "Static Analysis with Paragraph Vector for Malware Detection," in *Proc. 11th International Conference on Ubiquitous Information Management and Communication (IMCOM 2017)*, pp. 80–86, 2017. DOI: 10.1145/3022227.3022306.

[7] Y. Oyama, "Trends of anti-analysis operations of malwares observed in API call logs," *J. Comput. Virol. Hacking Tech.*, vol. 14, no. 1, pp. 69–85, 2018. DOI: 10.1007/s11416-017-0290-x.

[8] S. S. Chakkaravarthy, S. Dhamodaran, and V. Vijayakumar, "A Survey on Malware Analysis and Mitigation Techniques," *Comput. Sci. Rev.*, vol. 32, pp. 1–23, 2019. DOI: 10.1016/j.cosrev.2019.01.002.

[9] A. Souri and R. Hosseini, "A State-of-the-art Survey of Malware Detection Approaches using Data Mining Techniques," *Hum. centric Comput. Inf. Sci.*, vol. 8, no. 1, pp. 1–22, 2018. DOI: 10.1186/s13673-018-0125-x.

[10] M. Kalash, M. Rochan, N. Mohammed, N. D. B. Bruce, Y. Wang, and F. Iqbal, "Malware Classification with Deep Convolutional Neural Networks," in *Proc. 9th IEEE International Conference on New Technologies, Mobility and Security (NTMS 2018)*, pp. 1–5, 2018. DOI: 10.1109/NTMS.2018.8328749.

[11] J. Singh and J. Singh, "A survey on machine learning-based malware detection in executable files," *J. Syst. Archit.*, vol. 112, p. 101 861, 2021. DOI: 10.1016/j.sysarc.2020.101861. [Online]. Available: https://doi.org/10.1016/j.sysarc.2020.101861.

[12] R. Thomas, *Lief - library to instrument executable formats*, https://lief.quarkslab.com/, 2017.

[13] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *Proc .49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2019)*, IEEE, pp. 52–63, 2019. DOI: 10.1109/DSN.2019.00020.

[14] X. Ling, L. Wu, W. Deng, *et al.*, "Malgraph: Hierarchical graph neural networks for robust windows malware detection," in *Proc. Conference on Computer Communications (INFOCOM 2022)*, IEEE, pp. 1998–2007, 2022. DOI: 10.1109/INFOCOM48880.2022.9796786.

[15] B. Athiwaratkun and J. W. Stokes, "Malware classification with LSTM and GRU language models and a character-level CNN," in *Proc. 42nd IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2017)*, pp. 2482–2486, 2017. DOI: 10.1109/ICASSP.2017.7952603.

[16] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *Proc. 38th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2013)*, pp. 3422–3426, 2013. DOI: 10.1109/ICASSP.2013.6638293.

[17] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware classification with recurrent networks," in *Proc. 40th IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2015)*, pp. 1916–1920, 2015. DOI: 10.1109/ICASSP.2015.7178304.

[18] F. Cohen, "Computer viruses: Theory and experiments," *Comput. Secur.*, vol. 6, no. 1, pp. 22–35, 1987. DOI: 10.1016/0167-4048(87)90122-2.

[19] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *Proc. 22nd IEEE Symposium on Security and Privacy (SSP 2001)*, pp. 38–49, 2001. DOI: 10.1109/SECPRI.2001.924286.

[20] J. Z. Kolter and M. A. Maloof, "Learning to Detect and Classify Malicious Executables in the Wild," *J. Mach. Learn. Res.*, vol. 7, pp. 2721–2744, 2006. [Online]. Available: http://jmlr.org/papers/v7/kolter06a.html.

[21] J. Saxe and K. Berlin, "Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features," in *Proc. 10th IEEE International Conference on Malicious and Unwanted Software (MALWARE 2015)*, pp. 11–20, 2015. DOI: 10.1109/MALWARE.2015.7413680.

[22] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware detection by eating a whole EXE," in *Proc. 32nd AAAI Conference on Artificial Intelligence Workshop (AAAIW 2018)*, ser. AAAI Technical Report, vol. WS-18, pp. 268–276, 2018. [Online]. Available: https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422.

[23] H. S. Anderson and P. Roth, "EMBER: an open dataset for training static PE malware machine learning models," *CoRR*, vol. abs/1804.04637, 2018. arXiv: 1804.04637. [Online]. Available: http://arxiv.org/abs/1804.04637.

[24] R. E. Harang and E. M. Rudd, "SOREL-20M: A large scale benchmark dataset for malicious PE detection," *CoRR*, vol. abs/2012.07634, 2020. arXiv: 2012.07634. [Online]. Available: https://arxiv.org/abs/2012.07634.

[25] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "BODMAS: an open dataset for learning based temporal analysis of PE malware," in *Proc. IEEE 42nd Security and Privacy Workshops (SPW 2021)*, pp. 78–84, 2021. DOI: 10.1109/SPW53761.2021.00020.

[26] M. Z. Shafiq, S. M. Tabish, F. Mirza, and M. Farooq, "A Framework for Efficient Mining of Structural Information to Detect Zero-Day Malicious Portable Executables," *nexGIN RC Technical Report, TR-nexGINRC-2009-21*, 2009. [Online]. Available: http://nexginrc.org/Publications/pub_files/tr21-zubair.pdf.

[27] Z. E. Xu, G. Huang, K. Q. Weinberger, and A. X. Zheng, "Gradient Boosted Feature Selection," in *Proc. 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2014)*, pp. 522–531, 2014. DOI: 10.1145/2623330.2623635.

[28] E. Tuv, A. Borisov, G. C. Runger, and K. Torkkola, "Feature Selection with Ensembles, Artificial Variables, and Redundancy Elimination," *J. Mach. Learn. Res.*, vol. 10, pp. 1341–1366, 2009. [Online]. Available: https://dl.acm.org/citation.cfm?id=1755828.

[29] G. Ke, Q. Meng, T. Finley, *et al.*, "Lightgbm: A highly efficient gradient boosting decision tree," in *Proc. 31st Annual Conference on Neural Information Processing Systems (NIPS 2017)*, pp. 3146–3154, 2017. [Online]. Available: https://proceedings.neurips.cc/paper/2017/hash/6449f44a102fde848669bdd9eb6b76fa-Abstract.html.

[30] E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE header, malware detection with minimal domain knowledge," in *Proc. 10th ACM Workshop on Artificial Intelligence and Security (AISec@CCS 2017)*, pp. 121–132, 2017. DOI: 10.1145/3128572.3140442.

[31] Y. Chen, S. Wang, D. She, and S. Jana, "On Training Robust PDF Malware Classifiers," in *Proc. 29th USENIX Security Symposium (USENIX Security 2020)*, pp. 2343–2360, 2020. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/chen-yizheng.

[32] S. E. Coull and C. Gardner, "Activation Analysis of a Byte-Based Deep Neural Network for Malware Classification," in *40th IEEE Security and Privacy Workshops (SPW 2019)*, pp. 21–27, 2019. DOI: 10.1109/SPW.2019.00017.

[33] E. M. Rudd, F. N. Ducau, C. Wild, K. Berlin, and R. E. Harang, "ALOHA: auxiliary loss optimization for hypothesis augmentation," in *Proc. 28th USENIX Security Symposium (USENIX Security 2019)*, pp. 303–320, 2019. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/rudd.

[34] Y. Wang, Y. Sun, Z. Liu, S. E. Sarma, M. M. Bronstein, and J. M. Solomon, "Dynamic graph CNN for learning on point clouds," *ACM Trans. Graph.*, vol. 38, no. 5, 146:1–146:12, 2019. DOI: 10.1145/3326362.

[35] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" In *Proc. 7th International Conference on Learning Representations (ICLR 2019)*, OpenReview.net, 2019. [Online]. Available: https://openreview.net/forum?id=ryGs6iA5Km.

[36] A. Narayanan, M. Chandramohan, R. Venkatesan, L. Chen, Y. Liu, and S. Jaiswal, "graph2vec: Learning Distributed Representations of Graphs," *CoRR*, vol. abs/1707.05005, 2017. [Online]. Available: http://arxiv.org/abs/1707.05005.

[37] F. Sun, J. Hoffmann, V. Verma, and J. Tang, "Infograph: Unsupervised and semi-supervised graph-level representation learning via mutual information maximization," in *Proc. 8th International Conference on Learning Representations (ICLR 2020)*, pp. 1–16, 2020.

[38] L. Yang, W. Guo, Q. Hao, *et al.*, "CADE: Detecting and Explaining Concept Drift Samples for Security Applications," in *Proc. 30th USENIX Security Symposium (USENIX Security 2021)*, pp. 2327–2344, 2021. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/yang-limin.

[39] M. Dib, S. Torabi, E. Bou-Harb, N. Bouguila, and C. Assi, "Evoliot: A self-supervised contrastive learning framework for detecting and characterizing evolving iot malware variants," in *Proc. ASIA CCS '22: ACM Asia Conference on Computer and Communications Security*, pp. 452–466, 2022.

[40] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Unsupervised Graph Contrastive Learning with Data Augmentation for Malware Classification," in *Proc. 16th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2022)*, IARIA, 2022, pp. 41–47, ISBN: 978-1-68558-007-0. [Online]. Available: https://www.thinkmind.org/articles/securware_2022_1_70_30034.pdf.

[41] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, 6:1–6:42, 2012. DOI: 10.1145/2089125.2089126. [Online]. Available: https://doi.org/10.1145/2089125.2089126.

[42] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annual Computer Security Applications Conference (ACSAC 2007)*, IEEE Computer Society, pp. 421–430, 2007. DOI: 10.1109/ACSAC.2007.21. [Online]. Available: https://doi.org/10.1109/ACSAC.2007.21.

[43] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection Using Gradient Boosting Decision Trees with Customized Log Loss Function," in *Proc. 35th International Conference on Information Networking (ICOIN 2021)*, IEEE, 2021, pp. 273–278. DOI: 10.1109/ICOIN50884.2021.9333999.

[44] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection Using LightGBM With a Custom Logistic Loss Function," *IEEE Access*, vol. 10, pp. 47 792–47 804, 2022. DOI: 10.1109/ACCESS.2022.3171912.

[45] M. Terada, M. Akiyama, T. Matsuki, M. Hatada, and Y. Shinoda, "MWS Datasets for Anti-Malware Research -Contribution to the community and its challenges- (in Japanese)," *IPSJ SIG Technical Report*, vol. 2020-IFAT-139, no. 8, pp. 1–6, 2020.

[46] A. Test, *Malware Statistics & Trends Report | AV-TEST*, 2022. [Online]. Available: https://www.av-test.org/en/statistics/malware/.

[47] Y. Gao, Z. Lu, and Y. Luo, "Survey on malware anti-analysis," in *Proc. IEEE 5th International Conference on Intelligent Control and Information Processing*, pp. 270–275, 2014. DOI: 10.1109/ICICIP.2014.7010353.

[48] J. Zhang, "Machine Learning With Feature Selection Using Principal Component Analysis for Malware Detection: A Case Study," *CoRR*, vol. abs/1902.03639, 2019. arXiv: 1902.03639. [Online]. Available: http://arxiv.org/abs/1902.03639.

[49] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection using Attributed CFG Generated by Pre-trained Language Model with Graph Isomorphism Network," in *Proc. 46th IEEE Annual Computers, Software, and Applications Conferenc (COMPSAC 2022)*, IEEE, 2022, pp. 1495–1501. DOI: 10.1109/COMPSAC54236.2022.00237.

[50] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection by Control-Flow Graph Level Representation Learning With Graph Isomorphism Network," *IEEE Access*, vol. 10, pp. 111 830–111 841, 2022. DOI: 10.1109/ACCESS.2022.3215267.

[51] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang, and M. Zhou, "Minilm: Deep self-attention distillation for task-agnostic compression of pre-trained transformers," in *Proc. 34th Advances in Neural Information Processing Systems (NeurIPS 2020)*, vol. 33, pp. 5776–5788, 2020. [Online]. Available: https://proceedings.neurips.cc/paper/2020/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

[52] M. Togninalli, M. E. Ghisu, F. Llinares-López, B. Rieck, and K. M. Borgwardt, “Wasserstein weisfeiler-lehman graph kernels,” in *Proc. 33rd Advances in Neural Information Processing Systems (NeurIPS 2019)*, vol. 32, pp. 6436–6446, 2019. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/73fed7fd472e502d8908794430511f4d-Abstract.html.

[53] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proc. 9th International Joint Conference on Natural Language Processing (IJCNLP 2019)*, pp. 3980–3990, 2019. DOI: 10.18653/v1/D19-1410.

[54] A. Grover and J. Leskovec, “Node2vec: Scalable feature learning for networks,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 855–864, 2016.

[55] A. van den Oord, Y. Li, and O. Vinyals, “Representation learning with contrastive predictive coding,” *CoRR*, vol. abs/1807.03748, pp. 1–13, 2018.

[56] Y. You, T. Chen, Y. Sui, T. Chen, Z. Wang, and Y. Shen, “Graph contrastive learning with augmentations,” *CoRR*, vol. abs/2010.13902, pp. 1–12, 2020.

[57] N. Reimers and I. Gurevych, “Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks,” *CoRR*, vol. abs/1908.10084, pp. 1–11, 2019.

[58] Y. Zhu, Y. Xu, Q. Liu, and S. Wu, “An empirical study of graph contrastive learning,” *CoRR*, vol. abs/2109.01116, pp. 1–25, 2021. arXiv: 2109.01116. [Online]. Available: https://arxiv.org/abs/2109.01116.

Acknowledgements

First and foremost I am extremely grateful to my supervisors, Prof. Hajime Shimada, Prof. Yukiko Yamaguchi and Prof. Hirokazu Hasegawa for their invaluable advice, continuous support, and patience during my PhD study. I would also like to thank Prof. Tutomu Murase for his support on my study and daily life. Their immense knowledge and plentiful experience have encouraged me in all the time of my academic research and daily life. Besides my supervisors, I would like to thank the examiners of my thesis, Prof. Yuichi Kaji and Prof. Takahiro Katagiri for examining this thesis.

I would also like to express my sincere gratitude to the "Interdisciplinary Frontier Next-Generation Researcher Program of the Tokai Higher Education and Research System.", for providing me the opportunity to pursue a higher academic degree.

I would like to thank all my friends, especially Dr. Jiquan Xie and Zhenguo Hu. It is their kind help and support that have made my study and life in the Japan a wonderful time.

Finally, I would like to express my gratitude to my parents. Without their tremendous understanding and encouragement in the past few years, it would be impossible for me to complete my study.

List of Publications

Journals Articles

[1] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection Using LightGBM With a Custom Logistic Loss Function," *IEEE Access*, vol. 10, pp. 47 792–47 804, 2022. DOI: 10.1109/ACCESS.2022.3171912.

[2] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection by Control-Flow Graph Level Representation Learning With Graph Isomorphism Network," *IEEE Access*, vol. 10, pp. 111 830–111 841, 2022. DOI: 10.1109/ACCESS.2022.3215267.

International Conference Proceedings

[1] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Unsupervised Graph Contrastive Learning with Data Augmentation for Malware Classification," in *Proc. 16th International Conference on Emerging Security Information, Systems and Technologies (SECURWARE 2022)*, IARIA, 2022, pp. 41–47, ISBN: 978-1-68558-007-0. [Online]. Available: https://www.thinkmind.org/articles/securware_2022_1_70_30034.pdf.

[2] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection using Attributed CFG Generated by Pre-trained Language Model with Graph Isomorphism Network," in *Proc. 46th IEEE Annual Computers, Software, and Applications Conferenc (COMPSAC 2022)*, IEEE, 2022, pp. 1495–1501. DOI: 10.1109/COMPSAC54236.2022.00237.

[3] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Malware Detection Using Gradient Boosting Decision Trees with Customized Log Loss Function," in *Proc. 35th International Conference on Information Networking (ICOIN 2021)*, IEEE, 2021, pp. 273–278. DOI: 10.1109/ICOIN50884.2021.9333999.

Domestic Conference Proceedings

[1] Y. Gao, H. Hasegawa, Y. Yamaguchi, and H. Shimada, "Gradient Boosting Decision Tree Ensemble Learning for Malware Binary Classification," in *Proc. Computer Security Symposium (CCS 2020)*, 2020, pp. 589–595. [Online]. Available: https://cir.nii.ac.jp/crid/1050292572111463552?lang=en.