Malware Detection and Classification using Transformer-based Learning

MS (Research) Thesis

By

Fyse Nassar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
JULY 2021

Malware Detection and Classification using Transformer-based Learning

A THESIS

Submitted in fulfilment of the
requirements for the award of the degree
of

Master of Science (Research)

by

Fyse Nassar

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY INDORE
JULY 2021

INDIAN INSTITUTE OF TECHNOLOGY INDORE

CANDIDATE'S DECLARATION

I hereby certify that the work which is being presented in the thesis entitled **Malware Detection and Classification using Transformer-based Learning** in the fulfilment of the requirements for the award of the degree of **MASTER OF SCIENCE (RESEARCH)** and submitted in the **DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, Indian Institute of Technology Indore**, is an authentic record of my own work carried out during the time period from July 2019 to July 2021 under the supervision of Dr. Neminath Hubballi, Associate Professor, Indian Institute of Technology Indore, India.

The matter presented in this thesis has not been submitted by me for the award of any other degree of this or any other institute.

05/07/2021

Signature of the Student with date

Fyse Nassar

This is to certify that the above statement made by the candidate is correct to the best of my knowledge.

05-07-2021

Signature of the Thesis Supervisor with date

Dr. Neminath Hubballi

Fyse Nassar has successfully given his MS (Research) Oral Examination held on 22/10/2021

October 22, 2021
(Dr. Shanmugam Dhinakaran)

Signature of Chairperson (OEB)
Date:

Signature of Thesis Supervisor
Date: 22-Oct-2021

Signature of Convener DPGC
Date: 22-Oct-2021

Somnath Dey

Signature of Head of Department
Date: 22/10/2021

ACKNOWLEDGEMENTS

First and foremost, I would like to thank Almighty God, who has sustained me through these best and toughest years of my life. Without His immense blessings, life would not have been the same.

I would like to thank my supervisor **Dr. Neminath Hubballi**, who was a constant source of inspiration during my work. With his continuous guidance and research directions, this research work has been completed. His constant support and encouragement have motivated me to remain streamlined in my research work. I am also grateful to **Dr. Somnath Dey**, HOD of Computer Science and Engineering Department, for all his help and support.

I am thankful to **Dr. Bodhisatwa Mazumdar** and **Dr. Amogh C Umarikar**, my research progress committee members for taking out some valuable time to evaluate my progress all these years. Their valuable comments and suggestions helped me to improve my work at various stages.

My sincere acknowledgement and respect to **Prof. Neelesh Kumar Jain**, Director, Indian Institute of Technology Indore, for providing me with the opportunity to explore my research capabilities at the Indian Institute of Technology Indore.

A significant part of my gratitude goes to two incredible people I met here, **Anup** and **Saumya**, who have always been there for me through all my ups and downs. Saumya and Anup - *Thank you for everything.*

Finally, my deep and sincere gratitude to my **family** for their continuous and unparalleled love, help and support. I am forever indebted to my parents for giving me the opportunities and experiences that have made me who I am.

Fyse Nassar

In Loving Memory Of
My Grandfather
&
All Others
Lost Along The Way

ABSTRACT

Malware detection and classification assumes significance owing to the rapid propagation and proliferation of new variants. Grouping these variants into families with similar traits allows us to develop mitigation techniques that work for an entire family. Machine learning algorithms are used extensively for malware detection and classification tasks with selected features taken from executable files. Although these methods have shown good performance, the choice of features chosen constrain their ability to detect novel malware samples. To alleviate this limitation, recently, deep learning methods are used, which propose to automate the feature engineering task by extracting hidden semantic relationships between elements (raw bytes, opcodes, API calls, etc.) of the file.

In this thesis, we describe Transformer-based malware detection and family identification methods. Our first proposed method uses static analysis to extract opcode sequences from executable files, which are used to train a Transformer-based model for Windows malware detection and classification. We show that our proposed method can perform malware detection using only a short sequence of opcodes taken from the portable executable files.

A more sophisticated malware uses code obfuscation techniques or memory-based attacks to avoid detection. Such memory-based malware reside in the RAM to carry out their attacks. To detect the obfuscated malware, we use memory dumps obtained using dynamic analysis. We represent these memory dumps as images to be directly used as an input to a Transformer-based model. We also compare the detection and classification performance of Transformer-based model with a few conventional machine learning models. Our extensive experiments with different datasets demonstrate that our proposed techniques achieve better classification performance compared to recent methods. Malware is a threat not restricted to a single operating system. Android-based malware attacks have gained tremendous pace owing to the widespread use of mobile devices. Hence, we extend the previously mentioned technique that uses short sequences of opcodes to detect benign and malicious Android applications.

Contents

List of Figures

List of Tables

List of Abbreviations

ANN	Artificial Neural Networks
API	Application Programming Interface
APK	Android Application Package
APT	Advanced Persistent Threat
AUC	Area Under the Curve
BERT	Bidirectional Encoder Representations from Transformers
BiLSTM	Bidirectional Long Short Term Memory
BOVW	Bag of Visual Words
BRIEF	Binary Robust Independent Elementary Features
CNN	Convolutional Neural Network
CSV	Comma Separated Values
DBN	Deep Belief Network
DoS	Denial of Service
DDoS	Distributed Denial-of-service attack
DLL	Dynamic Link Libraries
DNS	Domain Name System
DoG	Difference of Gaussians
DPI	Deep Packet Inspection
DT	Decision Trees
FAST	Features from Accelerated Segment Test
FCG	Function Call Graph
FCN	Fully Convolutional Network

GELU	Gaussian Error Linear Units
GPU	Graphics Processing Unit
HOG	Histogram of Gradient
HTTP	Hypertext Transfer Protocol
kNN	k Nearest Neighbour
LSTM	Long Short Term Memory
MD5	Message-Digest Algorithm 5
ML	Machine Learning
MLP	Multilayer Perceptron
NB	Naive Bayes
ORB	Oriented Fast and Rotated BRIEF
OOA	Objective Oriented Association
PCA	Principal Component Analysis
PE	Portable Executable
RF	Random Forest
ReLU	Rectified Linear Units
Res-Net	Residual Networks
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
SHA-1	Secure Hash Algorithm 1
SIFT	Scale-Invariant Feature Transform
SMO	Sequential Minimal Optimization
SSL	Secure Socket Layer
SVM	Support Vector Machine
t-SNE	t-Distributed Stochastic Neighbour

UA User Agent

URL Uniform Resource Locator

ViT Vision Transformer

Chapter 1

Introduction

Malware is a piece of code written with malicious intention. It can cause disruptions in the operations of a computing and networked environment. These software are used for a variety of tasks like gaining complete control of a target system, harvesting sensitive information, click fraud, spamming, Denial of Service (DoS) and other such activities. Monetary benefits have overtaken the fun factor to become the impelling cause for an increase in diversity and sophistication of malware [1]. Given the implications of such infections, it is imperative that such software are detected before damage is done. Traditionally antivirus solutions are tasked to detect such software by having signatures for every type of malware. Signature generation is a manual effort and limited by expert knowledge. Moreover, these methods can not detect mutated or new variants of the malware. Of-late (particularly in the last decade), researchers have proposed different techniques adopting Machine learning algorithms [2] for detecting malware. These models use a feature set generated from executable files of known normal and/or malware samples using which machine learning algorithms are trained.

The malware detection methods broadly fall into three categories as below:

(i) Static Analysis: In this method of malware detection, the executable binary is analyzed without running it. Typically the executable is processed using a disassembler which reverse engineers the logic and generates an approximate code. Using this code, abnormal patterns or control flows are identified, which correspond to malware.

Such features can be used independently or in combination to detect the malware. Although these methods are simple, accuracy depends on the performance of the disassembler.

(ii) Dynamic Analysis: Here, the executable is run in a controlled (sandbox) environment to gain behavioural insights of executable. The behavioural observations can be payload installation, network traffic, CPU and memory usage patterns, API calls made, etc. Although dynamic analysis is more accurate compared to static analysis, it is notoriously time consuming and involves laborious manual effort.

(iii) Hybrid Methods: In these methods, a combination of both static and dynamic analysis is used to incorporate the best of both techniques. For a given malware, static parsing techniques are used to analyze it, whereas a dynamic analysis is performed on the same malware to observe the program's activity during execution. Such an approach intends to take advantage of the aforementioned techniques, which are speed and accuracy.

Rest of this chapter is organized as follows. We elaborate the motivation behind our work in Section 1.1. In Section 1.2, we present the summary of thesis contributions, and an outline of the rest of the thesis is described in Section 1.3.

1.1 Motivation

To stop malicious attacks on systems, malware needs to be detected at the earliest possible and be prevented from executing its code. Malware authors create different variants of existing malware to bypass detection. Such malware can be grouped into families as many of them share common traits such as codebase. Hence, identifying the family of a malware is an equally important problem as it helps to understand the previously used techniques by the attacker. Some of the existing static analysis techniques have used opcode sequence present in a program as an indicator of its maliciousness as opcodes are closely related to a program's logic. However, the number of opcodes present varies from file to file. Hence, it is necessary to rely on the opcode sequence of shorter lengths to reduce the computational complexity incurred in dealing

with sequences of larger lengths. Also a need arises to detect the more sophisticated malware that uses code obfuscation techniques to avoid detection. Different machine learning algorithms have been used in the literature, having varying performance [3]. Feature engineering is an important task, and it is laborious. It is not guaranteed that feature engineering done with a few sample sets of malware will work on future variants, requiring a new set of features. Deep learning is helpful in such cases to automate the task of feature engineering, hence we use it for malware detection and classification.

In this thesis, we aim to achieve the following objectives:

1. To develop a technique that uses short sequences of opcodes to detect and classify Windows-based malware.

2. To develop a technique that can detect and classify obfuscated Windows-based malware.

3. To develop a technique to detect Android-based malware using short sequence of opcodes.

1.2 Thesis Contribution

In this thesis, we propose techniques for malware detection and classification. The contributions of our thesis are as follows:

I. Windows Malware Detection and Classification Using Opcode Sequences: Existing methods in the literature that rely on static analysis use a file's entire opcode sequence or use n-grams of opcodes extracted from the executables. However, such an approach is computationally expensive. Thus as our first contribution, we show that Windows-based malware can be detected and classified (malware family identification) with high accuracy using only a few opcodes taken from an executable file. Our proposed method uses static analysis in which a disassembler is used to extract opcode sequences from executable files. These opcode sequences are used to train a

Transformer-based model. The proposed model can perform malware detection and identify its family using only a short opcode sequence extracted from a portable executable file. Further, by conducting ablation studies, we show the effects of opcode sequence length, tokenizers and number of encoder blocks on the performance of the proposed model. Our extensive experiments on two datasets show that our method outperforms other similar techniques found in the literature.

II. Windows Malware Detection and Classification Using Memory Dumps: For the detection of sophisticated malware that are obfuscated and carry out memory-based attacks, a dynamic analysis approach is needed. Hence, we propose a dynamic analysis based technique that relies on the memory dump corresponding to a program extracted from the RAM. These memory dumps corresponding to each file are converted into images. Doing so avoids the need to rely on expert's knowledge to extract features from the memory dumps to be used for detection. To detect and classify the generated images of the malware, we follow two different approaches. In the first approach, we use different hand-crafted features that are extracted from the images. These features are used to train different machine learning classifiers that can then detect the images that belong to malware samples and identify their families. Instead of relying on hand-crafted features, in an alternative approach, we use the images directly as an input to a Transformer-based model, ViT, for detection and classification. We evaluate the models on a publicly available dataset and show that the proposed approaches perform better than other similar techniques found in the literature.

III. Android Malware Detection: Malware is a threat not just to Windows-based systems but also to other operating systems such as Android. Hence, we propose a technique to detect Android-based malware. We extend the previously mentioned technique that used opcode sequences for Windows malware detection and classification to detect Android-based malware. We disassemble the Android applications having *.apk* (Android Application Package) format and extract the opcode sequences from it. These opcode sequences are used to train a Transformer-based model. This

model is used to detect the android malware using only a few opcodes extracted from it. We evaluate the performance of the model on a dataset containing benign and malicious Android applications.

1.3 Organization of Thesis

The rest of the thesis is organised as follows:

Chapter 2: In this chapter, we describe a static analysis based technique that used opcode sequences from executables for detection and classification of Windows-based malware.

Chapter 4: In this chapter, we describe a memory dump based based technique for detection and classification of Windows-based malware.

Chapter 5: In this chapter, we deal with Android Malware detection technique based on opcode sequences.

Chapter 6: In this last chapter, we present a summary of the work done in this thesis followed by possible future scope of our work.

Chapter 2

Literature Survey

Malware, short for **mal**icious soft**ware** is a general term used for software that is designed to interfere with the normal functioning of a computer. As mentioned in the previous chapter, it can perform a range of malicious activities, from theft of sensitive data to damaging entire systems or devices. Differentiating and studying different types of malware helps us in understanding how they can infect the systems, the level of harm they pose and how to safeguard against them. Depending on the purpose, researchers classify the malware into various categories that are not mutually exclusive. Some of the most common and significant malware types are:

- **Adware:** These are the programs that are developed to display ads on computers and other devices. They are also used to redirect search requests to other advertising websites and further collect user's data for marketing. It generates revenue for the attacker when a user clicks on such ads. A commonly seen example of adware while browsing is the pop-up box with the message *"your computer has been infected"* along with a listed number that the user is asked to call to purchase the software in order to remove the infection.

- **Bot:** These are automated programs that are designed to surreptitiously gain access to a computer and then carry out the instructions received from a remote command and control (C&C) server. Usually, many such infected systems together form a botnet which are collectively used to carry out attacks. One of the

most common forms of attacks using botnet is a DDoS attack which attempts to make a system or entire domain unavailable. They are also used to distribute other kinds of malware.

- **Ransomware:** These kind of malware when run, disables the functionality of the system in some way. The most common techniques are to encrypt the users' files or revoke user's access by locking the system. In order to regain the access, a ransom payment needs to be paid. This ransom payment is often demanded in the form of virtual currency so that the attacker's identity remains anonymous.

- **Rootkit:** A rootkit allows an attacker to establish command and control over a victim's computer without the victim's knowledge. Once the rootkit gets installed, the attacker gets the ability to remotely execute files and even modify the system configuration of the host machine by gaining administrator-level access.

- **Spyware:** These software are used to spy and collect sensitive information like passwords, credit card information through unauthorized access to a victim's computer. Such harvested information is sent to the attacker by the spyware program. A popular spyware is a keylogger which secretly maps and stores all the keystrokes entered by a user.

- **Trojan:** It is a type of malicious software that misrepresents itself as a benign software to persuade users to install it on their systems. Most of the ransomware attacks are carried out using them by carefully placing the malicious code inside an apparently benign data. Since they are disguised as legitimate software, they are considered to be one of the most dangerous malware.

- **Virus:** It is one of the oldest types of malware that operate by inserting or attaching itself to a legitimate program. They cannot run independently as they depend on the underlying host program to get activated and carry out their malicious activity.

- **Worm:** These programs are similar to viruses and exploit the vulnerabilities that exist in the operating system to steal or delete data. But unlike viruses,

worms do not require host activity or other files to self-replicate or get activated. They typically rely on a computer network for their propagation.

In order to detect the malware, different analysis approaches are used in the literature. A summary of these approaches is shown in Figure 2.1. We elaborate these approaches in the following sections.

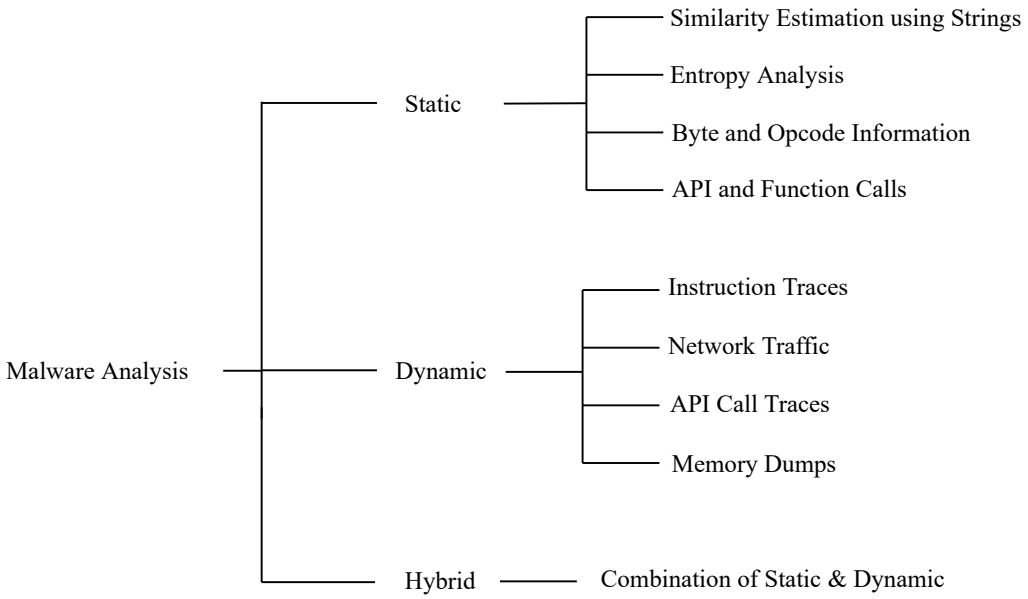


Figure 2.1: An Overview of Different Windows Malware Analysis Techniques

2.1 Static Analysis

In this method of analysis, the executable binary is examined without actually running it. Such an analysis attempts to understand the logic of the program by looking at its object code or structure. Further, it is also useful in generating a set of signatures which are used to uniquely identify the malware. One of the most common methods of signature generation is using hashing. The malware is run through a hashing program that generates a unique hash value used as its signature [4]. Some of the common hashing techniques are Message-Digest Algorithm 5 (MD5), Secure Hash Algorithm 1 (SHA-1). Depending upon the features, some of the common static analysis techniques used in the literature are explained below:

1. **Similarity Estimation using Strings**

   Strings extracted from files using tools are used for finding the similarity between files [5, 6]. These strings can contain information like URLs, domain names, IP addresses which the program intends to access. It can also contain information like the different Windows Dynamic Link Libraries (DLLs) that the program loads, registry keys etc. The basic idea is that if the strings obtained from a file under consideration is similar to those of a previously seen malware, then it is highly likely to be a mutation of that malware file [5, 6]. A similar idea was used by Konopiskỳ [7] who showed that computer generated string names for function names, method names and variables are more common in malicious files. They extract text strings such as function names, method names or variable names from a file. In such strings, if the ratio of the consonants to vowels is greater than a predetermined threshold value, the string is likely to be computer generated. Similarly, if the number of consonants in a sequence without a vowel is greater than a predetermined threshold value, then the string considered is likely a computer generated string. Similarity estimation using strings has also been used in conjunction with other features derived from static or dynamic analysis for an improved performance [8].

2. **Entropy Analysis**

   In order to hide the malicious content of a file, various obfuscation techniques are used by the malware authors. One of the popular techniques is packing in which encryption and compression of malicious code segments are done. Such code segments have higher entropy than the regular unobfuscated code. In order to detect such segments, entropy analysis is used. A high entropy value indicates that the considered piece of code segment consists of distinct values. This was shown in work done by Lyda and Hamrock [9] who observed that the average entropy for encrypted and compressed executables is 7.17, 6.80 respectively, whereas average entropy is only 5.09 for native executables.

   Building upon this idea, Sorokin [10] implemented a technique under the assump-

tion that the order of code and data areas is similar for variants of a malicious program. Files were split into different segments based on the different entropy levels calculated using wavelet analysis. A similarity score (edit distance) was computed between the respective segments of unknown files with known ones. Baysa et al. [11] studied the effect of structural entropy in metamorphic viruses and worms using the same method described in [10]. However, it was noted that the similarity score was sensitive to the file length as longer files tend to produce more segments.

3. **Byte and Opcode information**

N-grams are one of the most commonly used features for malware detection and classification. An n-gram is a set of co-occurring elements. These can be either bytes or opcodes. Byte n-grams are extracted from the binary file whereas opcode n-grams are extracted from assembly level instructions. Such instructions are obtained by reverse engineering the program by disassembly process, i.e. the machine code is translated into assembly level language. Table 2.1 shows a summary of previous works that used byte and opcode information for malware detection and classification.

Abou-Assaleh et al. [12] introduced a technique using n-grams of code and used K Nearest Neighbour(kNN) classifier for the detection of malicious opcodes. A representative profile was created for each benign and malicious class. A target file was compared to these profiles and matched to the closest one, then assigned its class label. Viruses and worms were the only two malware types present in their dataset.

Moskovitch et al. [13] carried out a study on opcodes using text categorisation concepts for the malware detection and classification task. N-grams of sizes 3 to 6 were extracted, and top features were chosen using Fisher score and classifiers like Artificial Neural Networks (ANN), Support Vector Machine (SVM), Naive Bayes (NB) and Decision Trees (DT) were trained. Moreover, an evaluation on an imbalanced dataset was performed where the percentage of malicious files

Table 2.1: Summary of Opcode and Byte Information based Works

Research Work	**Features**	**Techniques**
Abou-Assaleh et al. [12]	Opcode n-grams	kNN
Moskovitch et al. [13]	Opcode n-grams	ANN, SVM, NB, DT
Santos et al. [14]	Opcode n-grams	SVM
Ding and Zhu [15]	Opcode n-grams	DBN
Hu et al. [16]	Opcode n-grams	Clustering
Cakir and Dogdu [17]	Opcode Sequences	Gradient Boosted Trees
Sung et al. [18]	Opcode + API Call sequences	BiLSTM
Jain and Meena [19]	Byte n-grams	NB, IL, DT, AdaBoost, RF
Jang et al. [20]	Byte n-grams	Clustering
Raff et al. [21]	PE Header bytes	LSTM
Raff et al. [22]	Byte sequences	CNN
Nataraj et al. [23]	Grey-scale Images	kNN
Gibert et al. [24]	Grey-scale Images	CNN
Rezende et al. [25]	Grey-scale Images	ResNet-50

were varied in the training and test set. They observed that the performance of a classifier was best when the percentage of malicious files in the training and test set is the same. Similar to the previous work, Jain and Meena [19] used bytes n-gram features for detection of malware. In order to reduce the feature space, classwise document frequency was used, which selects the top k n-grams which are present in the maximum number of files belonging to a particular class. These features are then used to train different classifiers available in the WEKA tool [26].

Santos et al. [14] used a method that was based on the frequency of opcode sequences present in a file. Several machine learning classifiers were trained, and they arrived at the conclusion that a higher number of labelled samples were needed for improved performance. In order to overcome this limitation,

in their follow-up work, they proposed several other techniques like collective classification [27], single class learning [28] and semi-supervised learning [29]. Cakir and Dogdu [17] represented opcodes using Word2Vec embeddings [30] and a gradient boosting algorithm was used for classification.

Growing number of malware variants pose a challenge for manual feature engineering. This is not only time consuming but also tedious to perform. Hence, scalable models which can automate this task were developed [16, 20]. Jang et al. [20] proposed BitShred, which focused on malware comparison using byte sequences. Feature hashing was used to reduce the high dimensional feature space, and further clustering was used to group similar malware. However, this was susceptible to binary level obfuscation. Hu et al. [16] developed MutantX-S, which extracted n-gram features from opcode sequences. Further, to reduce the dimensionality of the extracted feature vectors, they used a hashing technique that lowered the computation and memory requirements. For clustering, the samples, instead of using the classical k means clustering algorithm, a prototype-based clustering that uses a small set of representative samples for faster computation was implemented. They show that their model is able to process more than 130,000 malware samples within a few hours.

Ding and Zhu [15] developed a model Deep Belief Network (DBN) whose building block is an autoencoder. Unlike the previous techniques, they used this deep learning model for reducing the dimensions of the input features. Further, n-gram vectors from unlabeled files can be used for unsupervised pretraining for effective encoding of the input vectors. They showed improved performance, in terms of accuracy, over baseline classifiers such as KNN, SVM and DT.

Since the computational complexity of generating n-grams is high, other alternatives were explored [21–23]. Raff et al. [21] showed that a fully connected and recurrent networks were useful in malware detection when trained on the first 300 bytes from the PE header of each file. They further extended the work by training networks on the entire byte sequence (several million bytes long) of

an executable [22]. Sung et al. [18] in their approach used opcode sequences combined with API function names which are then embedded using fastText algorithm.

An interesting approach to detect malware was proposed by Nataraj et al. [23], who represented malware as images. The binary files were represented as grey-scale images. They observed that samples belonging to the same family are visually similar and, at the same time, distinct from the samples of other families. This was primarily due to the usage of existing codes for creating variants. They extracted GIST features from the image, which are low dimensional and discriminative image vectors used for identification of image [31]. These features are used to train kNN with euclidean distance for the classification task. However, adversaries can obfuscate their malicious code to bypass this texture analysis technique which used global image-based features. Instead of using hand-crafted features and to automate feature extraction, Gibert et al. [24] trained a convolutional neural network using such grey-scale images for classification, whereas pre-trained weights of ResNet-50 architecture was used for the same [25].

4. **API and Function Calls**

Application Programming Interface (API) calls are used by programs to access the services of the operating system. API calls provide an abstraction to the task that needs to be performed, i.e. the caller need not know about how the underlying operation gets executed. Almost all the programs that run on Windows OS use Windows API calls to communicate with the OS. For example, to create a new file or to open an existing file, *OpenFileW* Windows API is used. Another example is, the set of API calls *WriteProcessMemory*, *LoadLibrary* and *CreateRemoteThread* are most likely used by malware for carrying out DLL injection into a process. Hence, API calls can reveal the behaviour of a program [32] and can be used for detecting the malware.

Zabidi et al. [33] performed a study on a few malware samples and found some API calls like *GetTickCount*, *GetModuleHandleW* etc. are exclusive to malware

samples. Sami et al. [34] implemented a framework to classify the Portable Executable file based on the usage of the API calls with the underlying idea that they can be used to know the behaviour of the executables. Initially, the files are processed using a tool called *PE analyzer*, and the list of imported API calls are extracted. Further, to extract only the discriminative features, the Fisher score was used. These features are then used for training a Random Forest model. Ye et al. [32] implemented a rule-based system. The API calls were extracted using a *PE parser* tool. Using these API calls, a static signature is generated, which is stored in a database. Class association rules are generated using Objective Oriented Association (OOA) algorithm. These rules, along with the API calls extracted from a target file, is used for detecting its maliciousness. Shankarapani et al. [35] studied the statistical properties of API calls in the files. They used the frequency of occurrence of API calls as an input feature for SVM, which was used for malware classification. Similar to the byte and opcode n-grams approach, Faruki et al. [36] showed that API calls n-grams are also good indicators for maliciousness of a file.

Kinable and Kostakis [37] approached the problem of malware classification using the function call graph (FCG) clustering technique. In such a graph, the vertices represent the functions in the program, whereas the function calls are represented by the edges. To construct such a graph, external tools like IDA Pro [38] or Radare2 [39] can be used. Graph matching technique was used to compute the pairwise graph similarity score, and further performance was evaluated using DBScan and K-medoids algorithms.

Due to the significant performance overhead incurred for graph comparisons in the previous work, Hassen and Chan [40] used a locality sensitive hashing (MinHash) to improve the speed of function similarity comparison. In addition to this, the graph was represented using vector representation with the help of MinHash signatures of the function. Saxe and Berlin [41] used different static features like PE import information, histogram of byte entropy features and

ASCII strings lengths and other meta-data from the static analysis. They use a deep learning model along with the Bayesian calibration approach, which instead of detecting malware in a binary sense, provided probabilities of the file being malicious.

The above proposed techniques are based on static analysis. Basically, in the static analysis, a file is examined without actually executing it. Reverse engineering the file by disassembly is a commonly used technique for analyzing the file. Although this method is simple, the accuracy depends on the performance of the disassembler used. Also, with the increasing sophistication of malware, many obfuscation techniques like code encryption, reordering the program instructions or packing of the executables are used to hide the real content of the malware and to thwart the analysis process. In order to overcome such hindrances, dynamic analysis methods are used by researchers, which is explained in the next subsection.

2.2 Dynamic Analysis

In this type of analysis, the program is allowed to run in a sandbox (controlled) environment to monitor its behaviour. This behaviour can be monitored at different levels starting from the lowest level possible (i.e. binary code) to the system as a whole (e.g. changes made to the file system or registry). Other behavioural observations include payload installation, network traffic, CPU and memory usage patterns, API calls made, etc. Unlike the static analysis, it shows the true flow of the actions taken by the program. Similar to the features obtained from static analysis, some of the features obtained from dynamic analysis used for malware detection in the literature are explained below:

1. **Instruction Traces**

   Unlike the sequence of instructions found in the assembly code of a file, the sequence of instructions executed when the file actually runs is different. This is due to the presence of function calls, jump statements and also other conditional

statements. Dynamic traces (sequence of processor instructions), hence, can overcome the hindrances due to obfuscation of code by encrypters and packers.

Carlin et al. [42] with the help of dynamic analysis, extracted instruction traces from both malicious and benign files. They performed two sets of experiments: i) Opcode count based detection using Random Forest, ii) a Hidden Markov Model to classify data using Opcode sequences. In their later work [43], they used opcode n-grams (1 to 3) for improved performance. O'Kane et al. [44] carried out work in determining the optimal set of opcodes necessary to detect the maliciousness of a file. They used opcode density histograms created using runtime traces and concluded that a decent detection rate is achievable using only a few opcodes. These opcodes were identified using Principal Component Analysis (PCA) [45]. Instead of using the computationally expensive n-grams, Anderson et al. [46] modelled the instructions traces as a Markov chain, where the vertices represented instructions and the transition probabilities are shown as the edge weights. Further, a similarity (kernel) matrix between the Markov chain graphs is constructed, which is then used to perform classification. A similar approach was used by Storlie et al. [47] who used a flexible spline logistic regression model for the detection of malware.

2. **Network Traffic**

   Malware generates network traffic for two reasons: to identify potential systems for infection/spreading; ii) to maintain connectivity with command and control servers by periodically exchanging data, to send/receive updates and commands etc. For such communications, the malware tries to use known network protocols to pass through firewalls. Hence, analysing the network activity of a program can give insights about the nature of the program.

   Kheir [48] carried out a study that explored User Agent (UA) anomalies within malicious HTTP traffic and extracts signatures for malware detection. Bekerman et al. [49] performed a study to detect suspicious data exchange with the Command and Control (C&C) servers. Their solution was based on cross-layers

(Transport, Internet and Application layers) and cross-protocol (DNS, HTTP, and SSL protocols) from which behavioural features were extracted. The obtained features were used to train different classification algorithms, out of which random forest worked the best.

In order to detect the Advanced Persistent Threat (APT) malware, Zhao et al. [50] carried out an analysis on the internet traffic and malicious DNS. They identified 14 features that can detect the APT malware C&C domains. Signature-based detection was also used along with anomaly detection that enabled a better defence to the system. However, malware infections that use the IP address directly instead of domains couldn't be detected using this approach.

Boukhtouta et al. [51] used Deep Packet Inspection (DPI) and flow packet headers for malware detection and classification. The malware was executed in a sandbox for three minutes to create a representative profile of malicious traffic. Further, bidirectional flow features like the number of forward and backward packets, packet size, minimum and maximum inter-arrival times for forward packets and other such features were used for different classification algorithms. A similar technique was adopted by Prasse et al. [52], who used LSTM based classifier, which is trained on the sequence of flows and detects whether the flow has its origins from a malware.

3. **API Call Traces**

Similar to instruction traces, the sequence of API calls that are used by a program depends on the execution environment. Such API calls are collected by executing the program, usually on a virtual machine. Analysing such a sequence of API calls can give an idea about what a program intends to do.

Rieck et al. [53] proposed a technique that used machine learning for automatic analysis of malware behaviour using information like API calls and its parameters. The files are executed in a sandbox environment, and such behavioural features were monitored. These observed behaviours were embedded into a vector space and further used by the machine learning algorithms. In order to

identify the novel classes of malware having similar behaviour, clustering was used. An unknown new malware is assigned to one of these classes using classification. This dual approach was used to handle an increasing number of files encountered on a day-to-day basis.

Uppal et al. [54] used n-grams of API calls as features for detection purpose. In order to select distinct and useful n-grams, an odds ratio of each n-gram is calculated, and further, the top-ranked features are chosen. An evaluation of 4 grams using SVM on a dataset of 270 binaries reportedly gave the best performance.

Dahl et al. [55] was the first to propose a deep learning model for malware classification using features obtained from dynamic analysis. They employ sparse binary features, including API trigrams, API calls with parameters, and file strings. Their architecture projected the high dimensional feature vector to a lower-dimensional dense vector. The authors found that increasing the number of hidden layers to 2 or 3 did not lead to an improvement in the accuracy when compared to a shallow architecture.

Huang and Stokes [56] performed manual feature engineering of API calls into 114 higher-level concepts. For example: Different ways to create a file include: i) *fopen()* call in C Language, ii) In user mode, *CreateFile()*, and iii) *ZwCreateFile()* method in kernel mode. All these different methods can be mapped to an abstract higher-level concept, *CreateFile* event. Along with this, API trigrams and null-terminated tokens are also used. The detection performance using a deep learning architecture was evaluated on a private dataset consisting of 6.5 million files, the largest in the literature.

The previous work [56] did not consider the order of invocation of the API calls. An alternative approach is to use the sequence of the API calls in their invocation order as features for the classifier. Galal et al. [57] in their work captured traces of API calls invoked by the files during its execution. Using a heuristic function, a representative semantic feature was created using the

extracted sequence of API calls. These semantic features provide high-level insight to a malware analyst. Athiwaratkun and Stokes [58] explored other deep learning architecture like LSTM to capture long term dependencies in API call sequences. Kolosnjaji et al. [59] combined both the recurrent and convolutional approaches on the API traces for optimising malware classification.

4. **Memory Dumps**

   Memory dumps refer to the volatile data that is extracted from a computer's physical memory. Since every program at some point during its execution ends up on the RAM, analysis of such memory dumps allows us to discover system inconsistencies that might indicate the presence of rootkit and also identify the overall state of the system [60]. Memory analysis is also useful in detecting fileless malware or in-memory malware, which leaves no trace on the disk.

   Mosli et al. [61] investigated three types of features extracted from memory images - API function calls, imported libraries and registry activities. Their results showed that registry activities gave the best performance for detection when used as input features to SVM. Case and Richard III [62] analysed memory dumps to detect Object-C malware. Javaheri and Hosseinzadeh [63] used multiple dumps taken at different times in order to detect obfuscated malware at the user and kernel level of OS. However, these methods using memory dumps are based on heuristic analysis and need deep knowledge of OS internals.

   Dai et al. [64] converted the memory dumps into grey-scale images using a similar approach in the earlier work [23], and image features such as Histogram of Gradient (HOG) is used to train multi-layer perceptron. Recently, Bozkir et al. [65] used RGB based encoding of the dump files and used global image features such as Gist [31] and HOG for classification.

While the main objective of dynamic analysis remains to detect the malicious activity of a program under execution, it is necessary to take additional precautions to avoid unnecessary risks to the underlying system. Two popular techniques for creating a risk-free environment are using a virtual machine or physical machines that

are set up with air-gapped networks, where the system is isolated from the internet or any other network. A drawback with the latter approach is that malware that runs on such machines have no network access. Hence, they may not be at their fullest capabilities due to no updates via the internet, command and control and other triggers. Using a virtual machine is seen as an alternative in such cases. With the increasing sophistication of some malware, they can detect the presence of a virtual environment and hence, show a benign behaviour to hinder the analysis process. Also, such techniques are not suitable for detecting malware that are dependent on some external event or waits for a specific time to trigger their malicious activities. Although dynamic analysis is more accurate compared to static analysis, it is notoriously time consuming and involves laborious manual effort.

2.3 Using Hybrid Approach

Hybrid analysis tries to overcome the limitations of the previous two techniques by combining both the techniques. A number of previous works used a combination of both static and dynamic features [66–68]. Pektaş and Acarman [66] used API call n-grams and other networking features for detection. Similar work was carried out by Islam et al. [67] where the features included API calls with their parameters and other string information.

Han et al. [68] built MalInsight that used a profiling approach where the features were built using: i) Basic structure profile that includes the size of PE sections ii) low-level profile, i.e. API and DLL information and iii) high-level profile, i.e. representative operations on files, registry and network. In their subsequent work [69], they studied the relationship between the static API calls and the dynamic API calls and built a semantic mapping between them. Their framework also gave insights for researchers regarding the malicious behaviour of a file.

Kolosnjaji et al. [70] built an ensemble of neural networks that has two sub-networks: i) a convolutional network that learns features from disassembled binaries ii) a feed-forward network that takes a set of features like PE Header metadata, list

of imported functions and DLL files. The outputs from each network are fed into a neural network that generates the final output.

In order to reduce the additional overhead incurred due to time for dynamic analysis, Kumar et al. [71] allowed a sample to be run only for four seconds. Behavioural information like network data, system calls, process and registry, were collected during this time period. Other static features from the PE header too were used for classification using different algorithms. A similar approach was followed by Rhode et al. [72] where the set of features that were considered is API calls and other system metrics. The system metrics included CPU usage, maximum process ID, bytes received and transmitted, memory use, swap use and the total number of processes.

Malware has its widespread attack not only on Windows-based systems but also on other operating systems like Android. Some of the recent works to detect Android-based malware are mentioned in the next section.

2.4 Android Malware Detection

Like Windows executables, features like opcodes, strings, etc., can be extracted from Android applications for malware detection. However, there are some features that are specific to Android applications that are indicators of its maliciousness. One such commonly used feature is the set of permissions that an app requires to execute specific code or functionality. Some of the android malware detection techniques are briefly explained below.

Vinayakumar et al. [73] proposed a permission based method to classify Android applications as benign or malicious. They used sequences of permissions as features for training LSTM, which is used to identify a malicious application. Nix and Zhang [74] used sequences of API calls extracted from the apps as features. With their extensive experiments, they show that their CNN model performed better than LSTM model for API call sequences.

Feizollah et al. [75] evaluated the effectiveness of using Android's intents for detecting malicious apps. An *intent* is a communication mechanism that allows the

usage of various functions provided by components of the same or other applications. They show that using intent based features can help in identifying malware better than using permission based features. However, they argued that using intent based features solely is not enough. Taheri et al. [76] took a hybrid analysis approach using a combination of features like permissions, intents, API calls, and network flows for android malware detection and family identification. These features were embedded and concatenated, further used to train a Random Forest classifier.

Similar to the earlier image based techniques used for detecting Windows-based malware, Al-Fawa'reh et al. [77] represented APK files as grey-scale images. Deep learning models like CNN with transfer-learning were used to detect the malware. Recently, Almahmoud et al. [78] compared conventional machine learning classifiers with deep learning model RNN for malware detection. The features used were a combination of permissions, permission rate, API calls and system events. They show that RNN outperformed the machine learning classifiers using these features. Jer [79] used n-grams of opcodes as features to detect and classify malware. They created a feature vector based on the presence or absence of an opcode n-gram. This was used to train an SVM classifier. McLaughlin et al. [80] used CNN on opcode sequences to avoid relying on manually designed malware features. Mateless et al. [81] proposed a technique to detect and classify malicious APK files based on their decompiled source code. They used NLP based methods on the decompiled code for malware detection.

2.5 Conclusion

With the growing amount of malware, manual analysis is no longer seen as a viable option. Hence, machine learning techniques are used in the literature to avoid manual analysis of the malware. Such techniques relied on features like opcodes, API calls, network traffic etc. Recent approaches have used deep learning models for malware detection, mainly due to their ability to handle large volumes of malware without relying on domain expert's knowledge to define discriminative features. It is not only important to detect malware but also to identify its family, as this helps in mitigating

the attacks by allowing one to quickly break down the techniques that the attacker had previously employed. Taking motivation from this, we describe Transformer-based models for the task of malware detection and classification.

Chapter 3

Windows Malware Detection and Classification using Opcode Sequences

3.1 Introduction

The increasing popularity of Microsoft Windows-based devices has led to a majority of the malware being created to target such devices. Many of the new malware are a variant of the existing malware. Hence, there arises a need to not only detect such malware but also in classifying them based on their family to mitigate their attacks. Although many of the existing works in the literature have tried to address this problem, we attempt to solve this problem by using Transformers, which offers superior classification accuracy in many other domains such as natural language processing [82] and computer vision [83]. Unlike the signature-based approach, this technique is successful against the unseen malware of the real world. In this chapter, we propose a Transformer-based model that uses opcode sequences for malware detection and its family identification.

In specific our contributions in this chapter are the following.

- We show that malware identification and classification can be done with high

accuracy using only a few opcodes taken from the beginning of an executable binary.

- We propose a Transformer-based model that can classify portable executable based malware samples using opcode sequences.
- We experiment with two different datasets and show that our method outperforms the other similar techniques found in the literature.

The remaining part of this chapter is divided into four sections. In Section 3.2 we summarise the closely related work in malware detection and classification using static analysis. In Section 3.3, we describe our proposed technique for malware detection and classification using opcode sequences. Experimental evaluation of the proposed technique is shown in Section 3.4. Finally we conclude this chapter in Section 3.5.

3.2 Related Work

As described in Chapter 2, several works [14, 16, 23, 41] have proposed techniques for malware detection using features obtained from static analysis. These prior works for malware detection use a significant amount of manual feature engineering, which requires domain level expertise. In order to reduce the amount of manual feature engineering, deep learning based techniques were explored in [17, 18, 21, 22]. The entire sequence of bytes in a file was used for detection in [22] whereas few other works [17, 18] use entire sequence of opcodes present in a file for detection. Instead of relying on a file's entire content, we propose a technique that uses only a few initial opcodes in a file for malware detection and classification, with minimal feature engineering.

3.3 Proposed Malware Classification

This section describes our proposed malware detection and family identification technique using a Transformer.

3.3.1 Problem Definition

Opcodes present in a program are closely related to a program's logic. Consider a file $f_i \in F$ where F is a collection of Windows OS binary executables and it is represented as $\langle OP_{f_i}, Class_{f_i} \rangle$ where OP_{f_i} represents the sequence of opcodes present in f_i. For malware detection, the term $Class_{f_i} \in (benign, malicious)$, while for malware classification, $Class_{f_i} \in (MF_1, MF_2, \cdots, MF_k)$ where MF_k denotes malware family k.

We define the problem of malware detection and malware family classification respectively as below:

- *Given a set of Windows binary executable test files F_T, we need to label them with a class identifier i.e. malicious or benign to each file $f_i \in F_T$ based on the opcode features OP_{f_i}.*
- *Given a test file collection F_T containing malicious files, we need to assign a malware family i.e. MF_k to each file f_i based on the opcode features OP_{f_i}.*

3.3.2 System Architecture

For malware detection and family identification, we use a deep learning framework that relies on the analysis of the opcodes extracted from the binary executables of Windows OS. It comprises of two main sub-components; a Feature Preprocessor and a Deep Learning based classifier. The framework is illustrated in Figure 3.1.

Feature Preprocessor: It consists of two components: (i) Disassembler, which generates the assembly level instructions of an executable input file; and (ii) Opcode parser, which takes the assembly-level code as input and gives the extracted opcodes as output.

Deep Learning based Classifier: From the extracted opcodes, a Transformer-based deep neural network is trained to perform the task of malware detection. In a similar fashion, a network is also trained for identifying the family of the given malware.

In the following subsections, we discuss the phases of opcode extraction and the use of deep learning based classifier in detail.

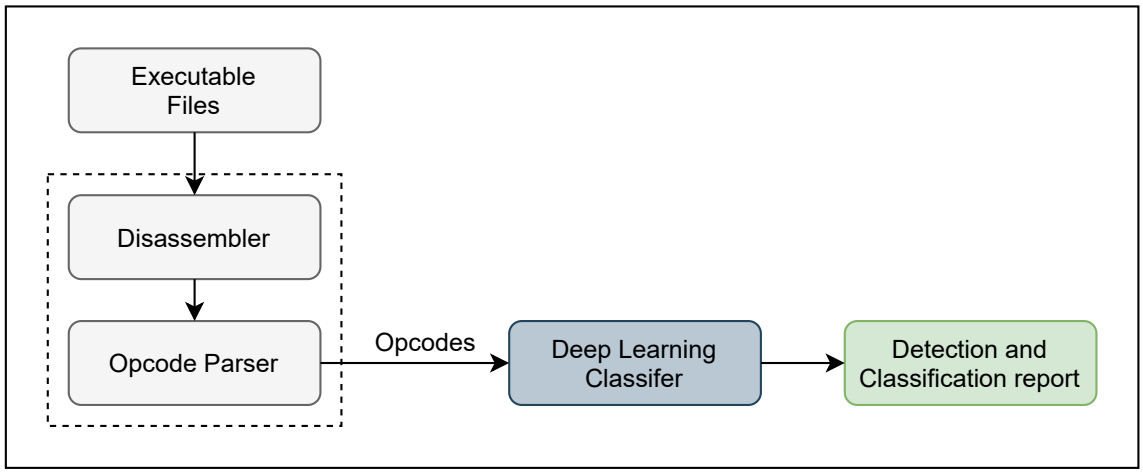


Figure 3.1: Proposed System Architecture

3.3.3 Opcode Extraction

In order to generate features for our classification model, we do a static analysis of the executable files. Specifically, we focus on the opcode sequences present in a file. To extract such opcodes, we use a disassembler. Given a file f_i, we initially disassemble the file using a disassembler which gives us an assembly file. This assembly file contains a series of addresses, instructions, API calls and other data. Figure 3.2 (a) shows a C function that calculates the square of a given number and Figure 3.2 (b) shows its corresponding assembly level code. The mapping between the two languages is shown by the different colour blocks.

```
1  int square (int num) {
2  return num * num
3  }
```

(a) A function to calculate square of a number in C language

```
square(int):
1  push   rbp
2  mov    rbp, rsp
3  mov    DWORD PTR [rbp-4], edi
4  mov    eax, DWORD PTR [rbp-4]
5  imul   eax, eax
6  pop    rbp
7  ret
```

(b) Corresponding assembly level code

Figure 3.2: A C function and its Corresponding Assembly Code

The instructions are generally represented in the form of opcode mnemonics fol-

lowed by its operands. For instance, in the instruction in line 2 of Figure 3.2 (b) *mov rbp*, *rsp*; *mov* is the mnemonic, *rbp* is the source operand and *rsp* is the destination operand. We will refer to the opcode mnemonics as just opcodes for the sake of convenience. We consider only the opcodes and ignore the operands as this representation is more robust against changes in the operands, as observed in a study carried out by Shabtai et al. [84]. Further, these extracted opcodes are represented by an opcode feature vector OP_{f_i}. Along with this, we also assign a class label $Class_{f_i}$ to the file where $Class_{f_i}$ is benign or malicious in case of malware detection and whereas for malware classification, it is the family name. Thus each file is represented by an opcode sequence, as shown below in Equation (3.1),

$$OP_{f_i} = \langle op_{i1},\, op_{i2},\, op_{i3}, \cdots, op_{ij}, \cdots op_{in} \rangle \tag{3.1}$$

where op_{ij} represents j^{th} opcode present in the file f_i having n opcodes. For the sample assembly code shown in Figure 3.2 (b), the opcode sequence would be of the form as shown in Equation (3.2). These sequences of opcodes extracted from all files of training set will be used to learn the relationships that exist between the opcodes using a Transformer-based model as elaborated in the next subsection.

$$OP_{f_{sample}} = \langle \cdots, push, mov, mov, mov, imul, pop, ret, \cdots \rangle \tag{3.2}$$

3.3.4 Deep Learning based Classifier

Since the input to the classifier is an opcode sequence, we use a Transformer [85] based architecture, which is effective in handling sequence based input. Unlike the conventional sequence-to-sequence architecture, which can only take short-term context into account, a Transformer-based architecture can encapsulate and preserve long-term dependencies. Moreover, the sequence-to-sequence architectures process sequence elements recursively, which hinders parallelisation while training. In contrast, the Transformer models refrain from recurrence, leading to a significantly reduced training time [86, 87].

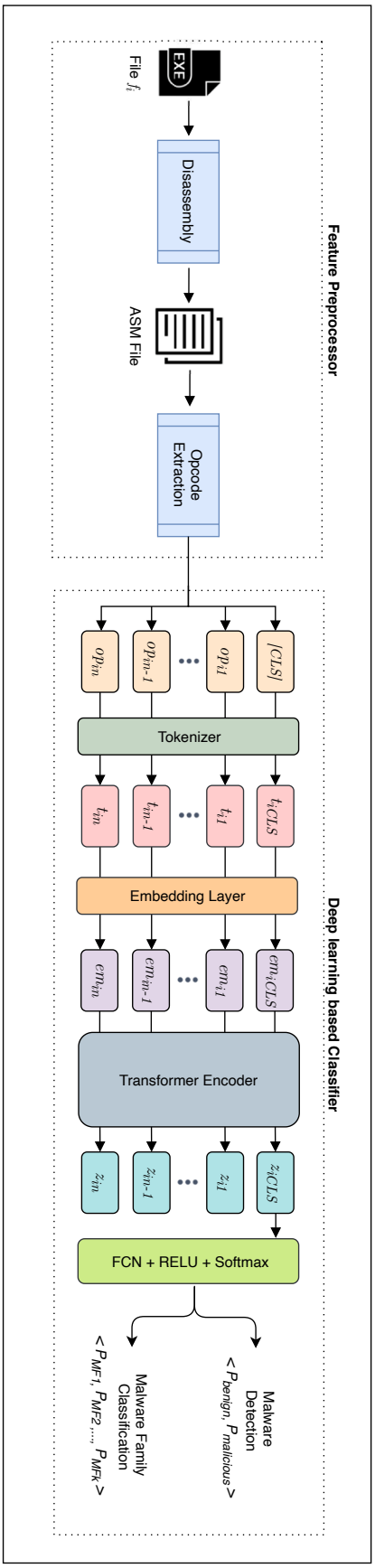


Figure 3.3: An Overview of the Proposed Approach

A Transformer architecture consists of two main components: an encoder and a decoder. We use a variant of the Transformer model called BERT [82] which uses only the encoders. Another key difference between BERT and a Transformer is that BERT can effectively capture information bidirectionally, that is from both the left and right context of a token (opcode), unlike Transformers which are unidirectional. Figure 3.3 shows an overview of the proposed classification approach. The extracted opcodes $OP_{f_i} = \langle op_{i1}, op_{i2}, op_{i3}, \cdots op_{in} \rangle$, are tokenized using a tokenizer algorithm to obtain tokens $T_{f_i} = \langle t_{i1}, t_{i2}, t_{i3}, \cdots t_{in} \rangle$ where $T_{f_i} \in \mathbb{N}$. Tokenization is essentially the process of splitting a sequence into smaller units, such as words or subwords. These smaller units are termed as tokens. These tokens are then passed through an embedding layer to obtain embeddings $EM_{f_i} = \langle em_{i1}, em_{i2}, em_{i3}, \cdots em_{in} \rangle$, where $EM_{f_i} \in \mathbb{R}^{n \times e}$, in which e denotes embedding dimension. In order to harness the global contextual information in a sequence, the self-attention technique is used which captures the relevance amongst every pair of tokens. To this end, each embedding EM_{f_i} is projected to Key K, Query Q and Value V matrices using three corresponding learnable weight matrices $W^K \in \mathbb{R}^{n \times e_k}$, $W^Q \in \mathbb{R}^{n \times e_q}$, and $W^V \in \mathbb{R}^{n \times e_v}$. For a given embedding EM_{f_i}, the self-attention $A \in \mathbb{R}^{n \times e_v}$ is calculated as product of the *values* to the softmax of the normalized dot product of *keys* and *queries* as shown in Equation (3.3).

$$A(Q, K, V) = \sigma \left(\frac{Q \cdot K^T}{\sqrt{e_q}} \right) V \tag{3.3}$$

where σ denotes the softmax operation.

For capturing multiple complex relationships amongst the tokens, the Transformer architecture computes multiple attentions for the given sequence. To achieve this, the architecture utilizes multiple self-attention blocks, each with its own set of learnable weights $\{W^{Q_b}, W^{K_b}, W^{V_b}\}$ where $b \in 0 \ldots B-1$ and B is the total number of attention blocks. For a given embedding Em_{f_i}, the output of B self-attention blocks are then concatenated into a single matrix $[A_0, A_1 \cdots A_{B-1}] \in \mathbb{R}^{n \times B \cdot e_v}$ which is then projected using the weight matrix $W \in \mathbb{R}^{B \cdot e_v \times e}$ to get the final output Z_{fi}.

There are two stages in training a BERT model: Pre-training, which is then followed by Fine-tuning. Pre-training task is usually performed in a self-supervised man-

ner on comparatively larger datasets, hence enabling the model in learning highly generalised and expressive relationships while circumventing cumbersome and expensive manual labelling [82, 85]. The model weights are then fine-tuned for the downstream tasks using a small-scaled dataset explicitly designed for the task at hand [88].

Pre-training: For pre-training stage, the model is fed with a sequence of opcodes, out of which 15% are masked. It is then trained to predict such opcodes based on the context. This technique is known as Masked Language Modelling [82]. An illustration of this concept is shown in Figure 3.4.

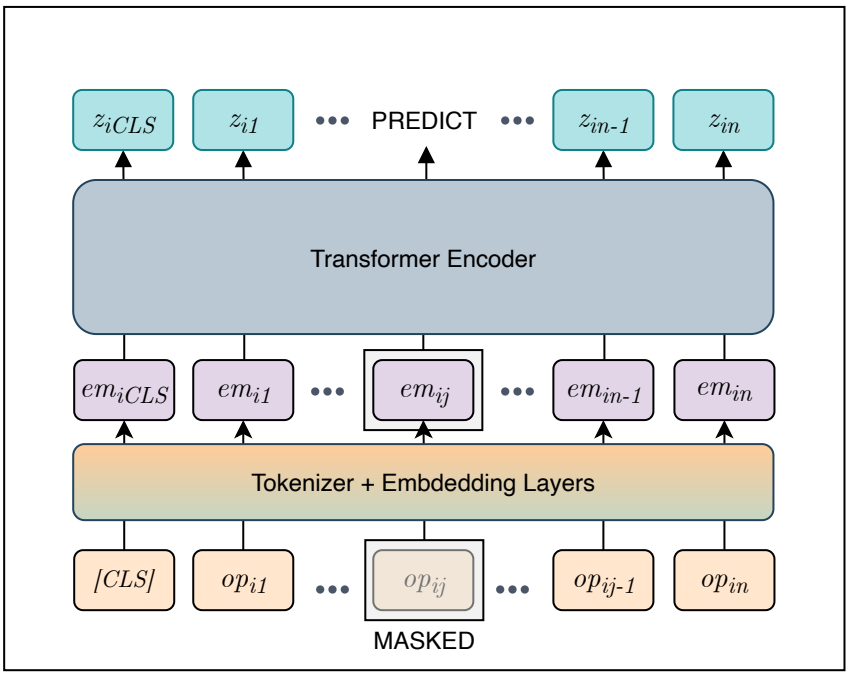


Figure 3.4: Illustration of the Pre-Training Task using Masked Language Modelling

Fine-tuning: For classification tasks, we make use of a special token named *[CLS]*. This token is prepended to the start of every sequence. Its corresponding output z_{iCLS} is given as an input to a fully connected layer as shown in Figure 3.5. Further, a probability distribution vector is generated using the softmax layer. This vector is of size 2 for the task of malware detection, where the classes are benign and malicious. Similarly, for the malware family classification task, this vector is of size K, representing the total number of distinct malware families. In the training phase, we compare this vector with the ground truth and loss is computed. This loss is then backpropagated

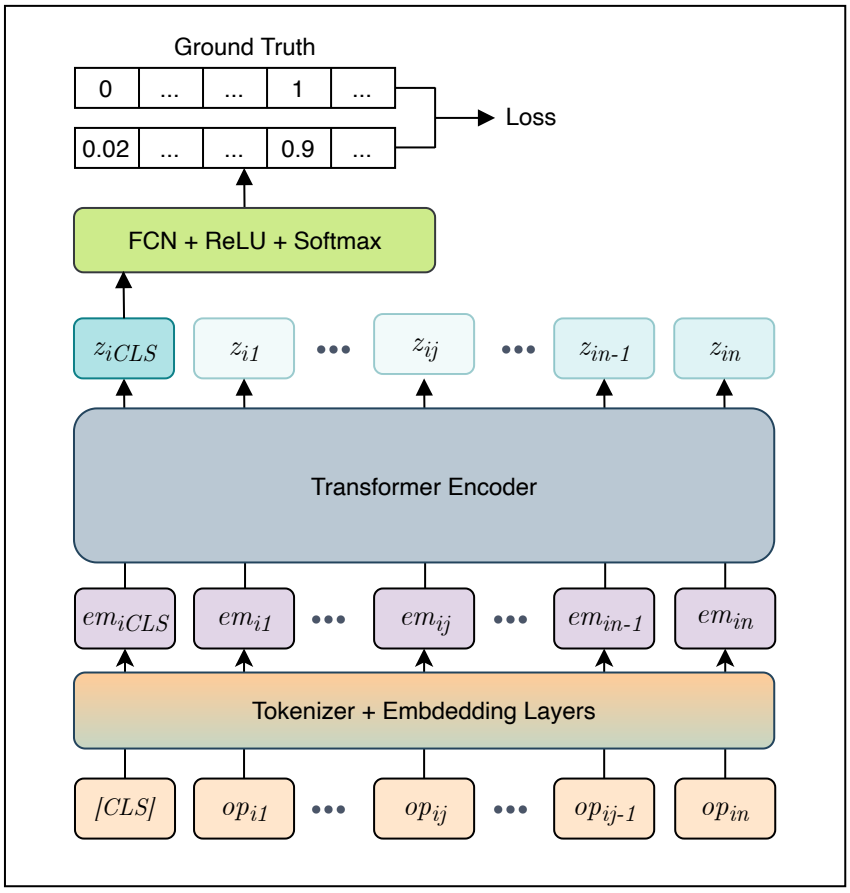


Figure 3.5: Fine-Tuning Task for Classification

to update the weights of the layers. Whereas in the testing phase, the predicted class is the one with the highest probability.

3.4 Experiments

In this section, we discuss the datasets used to evaluate our model. Subsequently, we explain the evaluation metrics used to calculate the efficacy of the proposed model. In the next subsection, we discuss the experimental setup and parameter selection. Finally, we present the results, followed by a performance comparison with previous works.

3.4.1 Dataset Details

The experiments are performed on two different datasets, both of which consists of Windows OS portable executable files. The first dataset contains benign and malicious files, while the second dataset consists of malware belonging to different families.

Dataset-I: This dataset has Microsoft Windows PE32 executable files marked as benign and malicious. We collected the benign files from a fresh installation of the Windows 8 operating system. Other open-source software were also added to this collection. Malicious files were collected from a repository of VirusTotal [89]. Table 3.1 shows the total the number of samples along with number of samples used for training and testing.

Table 3.1: Characteristics of Dataset-I

File Type	**Total Samples**	**Training**	**Testing**
Benign	4349	3044	1305
Malicious	5778	4044	1734
Total	10127	7088	3039

Dataset-II: Second dataset used in our experiments has disassembled code and corresponding byte codes of Microsoft Windows PE32 files of known malware families. These files are collected from the Microsoft Malware Classification challenge [90]. This malware family dataset (further referred to as Dataset-II) is publicly available on Kaggle [91] and has been widely used for research purposes. This dataset, when uncompressed, is half a terabyte in size and consists of nearly 10k labelled malware samples classified into 9 distinct malware families. Every sample is represented using two files. One is a byte file containing the malware's binary contents represented using a hexadecimal format, and the other is a disassembled file obtained using IDA Pro disassembler [38]. Table 3.2 presents the family distribution of the malware samples.

Table 3.2: Characteristics of Dataset-II

Family Name	**Total Samples**	**Training Samples**	**Testing Samples**	**Type**
Gatak	1012	708	304	Backdoor
Kelihos_ver1	386	270	116	Backdoor
Kelihos_ver3	2935	2054	881	Backdoor
Lollipop	2470	1729	741	Adware
Vundo	446	312	134	Trojan
Ramnit	1513	1059	454	Worm
Simda	33	23	10	Backdoor
Tracur	294	206	88	TrojanDownloader
Obfuscator.ACY	1168	818	350	Other Obfuscated malware
Total	**10257**	**7179**	**3078**	-

3.4.2 Evaluation Metrics

We use accuracy, precision, recall and F-1 score as metrics to evaluate the performance, and they are defined as follows:

1. **Accuracy**: It is calculated as the number of samples predicted correctly divided by the total number of samples.

$$Accuracy = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} \tag{3.4}$$

2. **Precision**: It is calculated as the number of samples which are actually malware divided by the total number of samples identified as malware.

$$Precision = \frac{T_P}{T_P + F_P} \tag{3.5}$$

3. **Recall**: It is calculated as the number of samples which are actually malware divided by the total number of malware samples.

$$Recall = \frac{T_P}{T_P + F_N} \tag{3.6}$$

4. **F1-score**: It is calculated as the harmonic mean of precision and recall.

$$F1Score = \frac{2 * Recall * Precision}{Recall + Precision} \tag{3.7}$$

where the total number of malicious and benign files correctly classified by the model is represented as True Positive (T_P) and True Negative (T_N) respectively. The total number of benign files which are misclassified as malicious by the model is represented as False Positive (F_P). Similarly, the total number of malicious files misclassified as benign by the model is represented as False Negative (F_N).

3.4.3 Experimental Setup

We use Dataset-I for the task of malware detection and Dataset-II for malware classification (family identification). For disassembling the executables, we use Ghidra disassembler [92], a collection of software reverse engineering tools developed by the National Security Agency, USA. We use a python script to extract the opcodes from the disassembled files. The extracted opcodes are then stored in a file with Comma-separated values (CSV) format, which is later used as an input to the model. In the case of Dataset-II, the files were already disassembled using IDA Pro [38]. Some of the files in this dataset contained no opcodes. Hence, we removed such files from both the training and testing sets. For our experiments, we split the dataset in a ratio of 70:30 for training and testing, respectively, as indicated in Table 3.1 and Table 3.2. We use the transformer library of Hugging Face [93] for the implementation of the BERT model. The batch size, which is the number of training samples that are processed in one iteration, is set to 32. The model weights are updated using *AdamW* optimiser [94]. The learning rate *(lr)* is set to 9×10^{-5}. Furthermore, we use cross-entropy loss and the number of epochs as 10 for training the model.

3.4.4 Ablation Study

In this section, we discuss the outcomes of the performed ablation study, which are done to select the hyper-parameters and techniques providing the optimum results. We do the ablation study on the following hyper-parameters and techniques:

1. **Tokenizer**: The BERT model uses the WordPiece [95], a subword segmentation algorithm for tokenization [82]. In the WordPiece algorithm, the vocabulary is

initialized to include every character and symbol present in the training dataset. Then the most likely combinations of symbols or subwords in the training set are added iteratively to the vocabulary until a predefined vocabulary limit is reached. The algorithm results in the words of the training set being split into one or more subword tokens. This approach has major drawbacks in our case due to a comparatively smaller vocabulary, such as opcodes present in Intel's instruction set. To explain this, consider an example of an opcode sequence $\langle mov, cmpsb \rangle$ having length 2. Using the WordPiece algorithm, the tokenized form of this sequence is $\langle mov, cmp, \#\#sb \rangle$ where ## is used as continuation symbol. This technique of tokenization changes the meaning of the opcode and also increases the number of tokens. To overcome this issue, we use word-level tokenization, where the tokens are generated using the space as a delimiter, thereby preserving the original form of the opcodes. This further reduces the complexity of the classification as the model no longer needs to learn to associate a large number of subword tokens to each label [96].

2. **Opcode Sequence Length**: A simple method is to use the entire opcode sequence of an executable for training and testing purposes. There exist several works in the literature which utilize this technique. However, the opcode sequence length varies from file to file. Furthermore, it is computationally expensive to train and test using the entire opcode sequence.

Table 3.3: Malware Detection Performance with Varying Sequence Length

Sequence Length	**Accuracy**
128	0.621
256	0.761
384	0.897
512	0.982

We study the sensitivity of the proposed method with respect to the sequence length for the classification task. The experiments are performed by changing

the length of the opcode sequence in steps from 128 to 512 and then testing it on Dataset-I. It can be observed from Table 3.3 that there is an improvement in model accuracy when we increase the length. The accuracy of the classification is at an acceptable level for a sequence length of 512. We consider this length of the opcode in our subsequent experiments.

3. **Number of Encoder layers**: The number of encoder layers in Base and Large variants of BERT is 12 and 24, respectively [82]. The number of parameters of the model is directly proportional to the number of encoder layers shown in Table 3.4. The Base and Large variants of BERT consist of massive 86 million and 171 million parameters.

Table 3.4: Number of Parameters in the Model on Varying the Number of Encoder Blocks

Number of Encoder Blocks	**Parameter Count (in millions)**
1	8.46
3	22.64
6	43.90
12	86.42
24	171.48

Sanh et al. [97] show that one can achieve comparable performance even with a significantly lower number of encoder layers and model parameters. They show that even by reducing the size of the BERT model by a margin of 40%, the model was able to preserve 97% of its understanding while improving the inference speed by 60%. Moreover, the previous experiments performed by Si et al. [98] and Li et al. [99] show that when working on a smaller vocabulary, optimum results are obtained for a lower number of layers. We observe that we can achieve on par performance by using only one encoder with only 8 million parameters. This is a significant reduction of 90% and 95% in models parameters,

respectively, compared to that of Base and Large variants of BERT. Moreover, we also note a significant decrease in the training and inference times when we use only one encoder layer.

3.4.5 Evaluation Results

We perform our experiments with settings mentioned previously to assess the performance of the proposed method for malware detection, using Dataset-I, and for malware classification, using Dataset-II. Table 3.5 and Table 3.6 show the outcome of evaluation using confusion matrices for these two cases respectively. It can be observed from Table 3.5 that there are 27 false malicious (identified as malware) out of 1305 benign executable files. Similarly, 27 out of 1734 malware samples are identified as benign. This further establishes that our model can efficiently detect both malware and benign files with minimal errors.

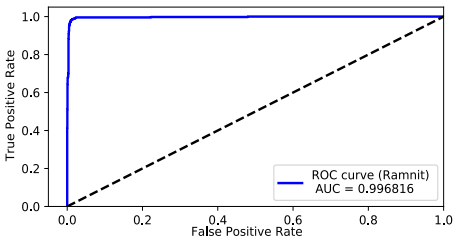
Table 3.5: Confusion Matrix for Malware Detection Experiment with Dataset-I

		Predicted	
		Malware	Benign
Actual	Malware	1707	27
	Benign	27	1278

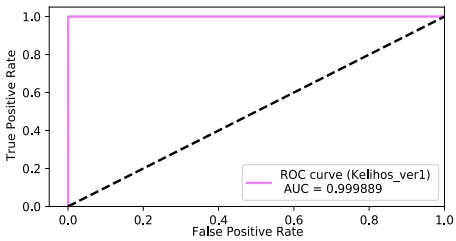
The confusion matrix of Table 3.6 shows the number of correct (entries of principle diagonal) and wrong predictions for experiments with Dataset-II. We can see that our model performs well in classifying the family of a given malware with a small number of misclassifications. It is worth noting that some of the previous studies [17] have used equal samples from only four families, whereas we consider all the families with an imbalance in the dataset. Thus our proposed model performs better with imbalanced sample sizes too. We also show the Receiver Operating Characteristic (ROC) curves for malware classification experiments in Figure 3.6. In Figure 3.6, for each subplot (a)-

Table 3.6: Confusion Matrix for Malware Classification with Dataset-II

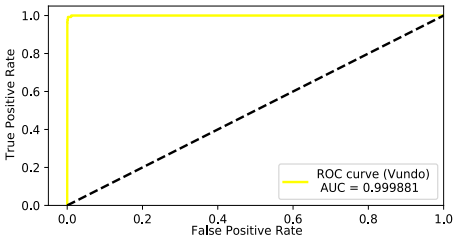
Actual \ Predicted	Ramnit	Lollipop	Kelihos_ver3	Vundo	Simda	Tracur	Kelihos_ver1	Obfuscator.ACY	Gatak
Ramnit	437	7	0	0	0	3	1	5	1
Lollipop	2	735	0	0	0	0	0	3	1
Kelihos_ver3	0	0	881	0	0	0	0	0	0
Vundo	0	0	1	132	0	1	0	0	0
Simda	0	0	0	1	7	1	0	1	0
Tracur	1	0	0	1	0	83	0	3	0
Kelihos_ver1	0	0	0	0	0	0	116	0	0
Obfuscator.ACY	13	1	0	3	0	0	0	330	3
Gatak	1	0	0	1	0	0	0	1	301



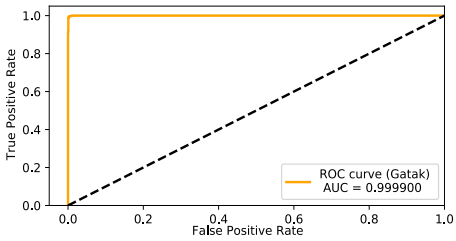
(a) Ramnit vs the others



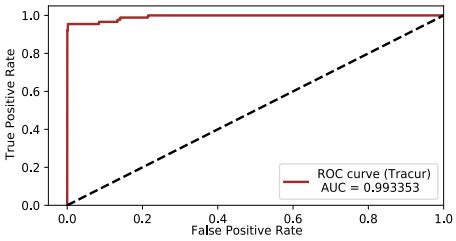
(b) Kelihos_ver1 vs the others



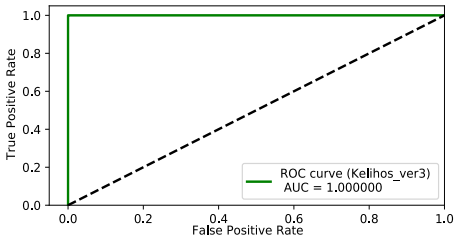
(c) Vundo vs the others



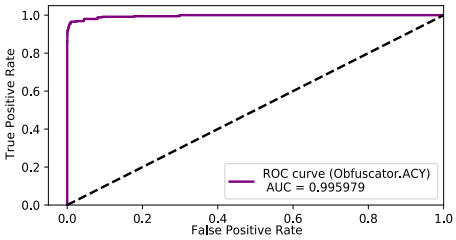
(d) Gatka vs the others



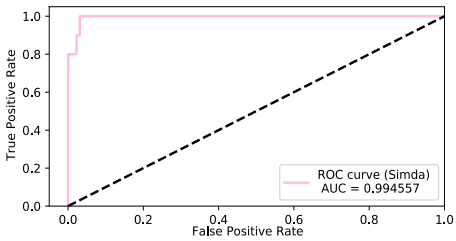
(e) Tracur vs the others



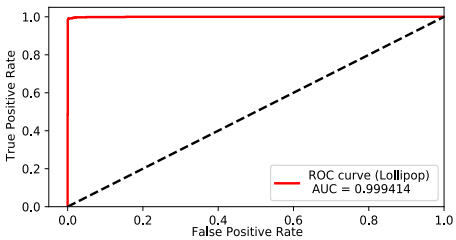
(f) Kelihos_ver1 vs the others



(g) Obfuscator.ACY vs the others



(h) Simda vs the others



(i) Lollipop vs the others

Figure 3.6: ROC Curves of Every Malware Family in Comparison to the Rest of the Families

(i), we consider one family as the positive class, whereas the rest of them collectively is considered as a negative class. The lowest AUC value is 0.994557 which is for the malware family *Simda* shown in Figure 3.6(h) and highest value of 1.0 for malware family *Kelihos_ver3* as in Figure 3.6(f).

3.4.6 Performance Comparison

For a more thorough analysis, we compare the performance of our proposed model for malware detection and classification tasks with the two recently proposed methods [17, 18].

Cakir and Dogdu [17] used a shallow deep learning based feature representation technique known as Word2Vec [30]. Word2Vec was originally used for generating embeddings for words in natural language. They trained a Word2Vec model using the opcodes sequences, which is then used to generate embeddings for individual opcodes. Opcodes having similar meanings have embeddings that are closer to each other in the vector space. These embeddings are given as input features to a Gradient Boosting Method [100], an ensemble technique, which is then used for malware family identification.

Sung et al. [18] in their approach augmented the static opcode sequences with API function names. They use a fastText based model which considers each opcode as a set of character n-grams which is then used for generating input embeddings that have a lower dimension than the traditional one-hot encoding approach. These input embeddings are fed into a Bidirectional LSTM network to analyse the correlation with sequential opcodes. Using the details mentioned in their paper, we have performed the experiments using their method and presented the results.

Both techniques have been used only for the classification of the malware family. We extend those techniques for both malware detection and family classification tasks. The performance comparison for malware detection on Dataset-I is shown in Table 3.7. It can be observed from Table 3.7 that our method has better accuracy, precision, recall and F-1 score.

Similarly, Table 3.8 provides the accuracy, precision, recall and F-1 score for the

Table 3.7: Performance Comparison for Malware Detection task on Dataset-I

Method	Sequence Length	Accuracy	Precision	Recall	F1-score
Cakir and Dogdu [17]	Entire sequence	0.952	0.945	0.920	0.932
Sung et al. [18]	Entire sequence	0.968	0.985	0.958	0.971
Proposed	512	0.982	0.984	0.984	0.984

experiments performed on Dataset-II for malware family identification. We can notice that our proposed approach outperforms these two methods in this case too.

Table 3.8: Performance Comparison for Malware Classification task on Dataset-II

Method	Sequence Length	Accuracy	Precision	Recall	F1-score
Cakir and Dogdu [17]	Entire sequence	0.955	0.957	0.955	0.956
Sung et al. [18]	Entire sequence	0.942	0.950	0.947	0.946
Proposed	512	0.981	0.981	0.981	0.981

3.5 Conclusion

Detecting malware and family identification is an important problem. In this chapter, we proposed a deep learning based malware detection method that uses opcode sequence from the Windows executable files. We use a Transformer-based model to learn semantic relationships between opcodes within a sequence. Our experimental evaluation with two datasets showed improved performance over the recent techniques. Further, our proposed model can work with initial few opcodes from executable files for detection and classification. Our current work has a limitation in the case of executable files which are encrypted or packed. We present techniques to handle such files in the next chapter.

Chapter 4

Windows Malware Detection and Classification using Memory Dumps

4.1 Introduction

In the previous chapter, we proposed a Transformer-based model that used opcode sequences extracted from a file using static analysis. However, such a technique would fail in two scenarios: i) when the code is obfuscated or encrypted ii) when there is no executable available for disassembly, i.e. the malware resides in the memory to carry out the attacks, also known as Fileless malware attacks [101]. Unlike traditional malware, such malware isn't written to disk and attacks the system without leaving any trace (file) due to it being present in RAM. One way to carry out fileless malware attacks is through phishing links on the internet when clicked, loads the malicious script into the RAM and accesses user's data [102].

In this chapter, we use memory based analysis through which memory dumps are extracted and further used for the detection and classification of malware. Our contributions in this chapter are:

- We perform a memory dump based malware detection and classification ap-

proach that can detect obfuscated and packed malware.

- We propose machine learning based models that uses features extracted from the memory dump images for detection and classification of malware.
- We also propose a Transformer-based model that can directly use the memory dump images for malware detection and its family identification.
- Our extensive experiment with a publicly available dataset shows that our method outperforms the similar techniques found in the literature.

The remaining part of this chapter is structured into four sections. In Section 4.2 we summarize the closely related work in malware detection and classification using memory dumps. In Section 4.3, we describe our proposed technique for malware detection and classification using memory dumps. Experimental evaluation of the proposed technique is presented in Section 4.4. In the end, we conclude this chapter in Section 4.5.

4.2 Related Work

As described in Chapter 2, some of the proposed techniques rely on manual feature extraction from memory dump [61, 63]. An alternative approach is to represent the memory dumps as visual images and use features such as image descriptors extracted from images for detection and classification [64, 65]. We perform a comparative analysis on different handcrafted image descriptors, which are used to train different machine learning algorithms. Also, we use a Transformer-based model for the detection and classification of the malware using the memory dumps.

4.3 Proposed Work

This section describes our proposed malware detection and classification technique using memory dumps. Our proposed approach has two main phases, which are elaborated in the subsequent subsections.

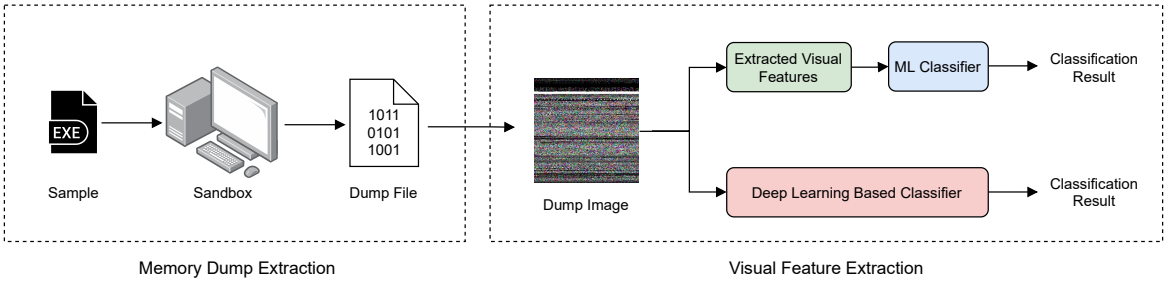


Figure 4.1: An Overview of the Proposed Approach

4.3.1 Memory Dumps

Any obfuscated or encrypted malicious file that executes on a system needs to decrypt itself in order to run. Such a program under execution will have its data in the memory. These data may include different information like text segments, data segments, layout of the thread stacks, heap areas, DLL calls of the process etc., which are captured using memory dumps. Memory dumps are mainly of two different types: i) full memory dump, where all the contents of system memory are recorded, ii) process memory dump, which extracts the contents specific to a process. The process specific memory dump file is considered for our work. The memory dump extraction and image representation phase are performed in a similar way as in the previous work [65]. We briefly explain it here.

Memory Dump Extraction: To obtain the dump files, the program is executed in a sandbox environment so as to avoid any harm to the underlying system. The program is allowed to run, and the memory dump is captured using a memory dump tool like *Procdump*[103]. A delay is added between the start of the program's execution and capturing of the dump in order to ensure that the program is active. Figure 4.1 shows the memory dump extraction phase.

Memory Dump Image Representation: The extracted dump files are converted into images so as to be used for different machine learning algorithm. These files are binary files consisting of 0s and 1s. Every byte has an unsigned value ranging from 0 to 255. For creating individual pixels of the image, we consider three consecutive bytes whose values correspond to Red, Green and Blue values of a pixel, respectively.

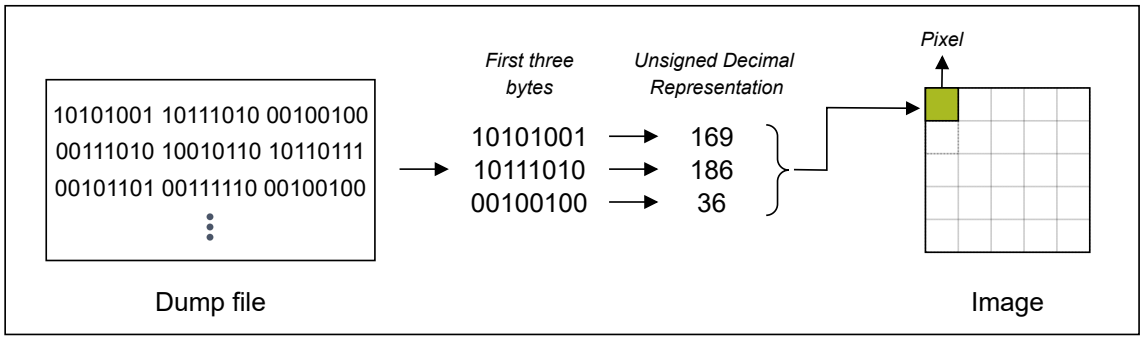


Figure 4.2: An Example of Converting Dump File to Image

Figure 4.2 shows a visualisation of converting a dump file to an image. Since the size of the dump file is dependent on the program it belongs to, the images will have varying sizes. Having a non-uniform and large-sized image can affect the efficiency of different feature extraction and classification algorithms. Hence, the images are transformed in a square sized shape using Lanczos interpolation available in the OpenCV library [104].

4.3.2 Visual Feature Extraction and Classification

As we are dealing with images, we need to extract image-based features that can be used for the detection and classification of the obtained memory dump images. The image-based features are generally divided into local and global features. The global features capture information from the whole image, whereas the local features give information about different patches of a given image. This is done by detecting multiple distinct keypoints in the image and by generating appropriate descriptors. Hence, unlike global features, which correspond to a single vector (descriptor), local features for an image correspond to multiple vectors (descriptors). Aly et al. [105] showed that the local feature descriptors perform better compared to global feature descriptors for image family classification. Hence, unlike the previous works that used global features, we explore the effectiveness of using local features in our work. Some of the local features that are extracted from the memory dump images are explained briefly below.

- **Scale-Invariant Feature Transform (SIFT) [106]:** SIFT performs keypoint detection, using Difference-of-Gaussians (DoG) operator. This operation is performed at various scales of the input image, hence making the detection process scale-invariant. The detected keypoints are then filtered, and unstable points are removed. Once the keypoints are detected, histograms of gradient orientations are computed around these points. The obtained values are normalized to between one and zero. The normalized values are the descriptors of the input image. The normalization helps the algorithm to achieve robustness against distortion, contrast and change in illumination. The size of the description vector obtained by SIFT is 128.

- **Oriented Fast and Rotated BRIEF (ORB) [107]:** ORB algorithm is a combination of Features from Accelerated Segment Test (FAST) [108] detection and rotation normalized Binary Robust Independent Elementary Features (BRIEF) [109] description methods. This algorithm generates a binary feature based on the FAST keypoint detector algorithm to detect interesting points from the input image. Further, top n quality points among these keypoints are selected using the Harris Corner algorithm [110]. Since the original BRIEF descriptor is susceptible to failure with rotation operation, a modified BRIEF based description algorithm is used. Due to this modified algorithm, ORB descriptors are invariant to rotation as well scaling and affine changes. The length of each description vector is 32.

- **KAZE Local Features [111]:** This algorithm operates in nonlinear scale-spaces, as opposed to SIFT, which works in Gaussian scale space. It works at the image's original resolution, unlike SIFT, which requires downsampling at each new octave. For detecting the keypoints, the Hessian matrix determinant is calculated and normalized at different scale levels. KAZE introduces rotation invariance by selecting a dominant orientation in a circular neighbourhood around each feature. The descriptors generated by KAZE are also invariant to limited affine and scale but is slightly more time expensive. Every detected

keypoint is described using a descriptor of length 64.

Using the above algorithms, for each image, a varying number of local descriptors are generated. As many machine learning classifiers require a fixed-sized feature vector as input, directly using the above generated local descriptors is not possible. Therefore we use a technique known as Bag of Visual Words (BOVW) representation which is used to convert these descriptors into a fixed-size feature vector. The BOVW consists of the following steps:

1. **Descriptor Computation:** In this step, the image keypoints are detected, and the corresponding descriptors are constructed using the above mentioned local descriptor extraction algorithm.

2. **Vocabulary Construction:** From the extracted descriptors (vectors), k-means clustering algorithm is used to cluster the feature vectors into k clusters. The centroids, i.e. cluster centres obtained, is considered as our visual vocabulary.

3. **Vector Quantization:** In this step, the description vectors of the input image obtained in step 1 are represented using a single feature vector. In order to do so, for each description vector, we find its nearest cluster centre from the vocabulary constructed in step 2. A histogram of length k is built where k is the number of centroids obtained. The i^{th} value of the histogram denotes the number of the descriptors whose nearest neighbour is centroid i. This process is known as vector quantization.

During the training, local descriptors from all the images in the training set are extracted. Using these descriptors, the vocabulary is constructed, and centroids are obtained. These centroids are used to quantize the descriptors to a single feature vector for each image in the training set. The feature vectors are then used for training different machine learning classifiers. Note that the vocabulary construction is performed only during the training phase and not during the inference or testing phase.

Figure 4.3 shows an overview of the steps performed during testing. During the time of inference, the image descriptors of test input are obtained. These descriptors

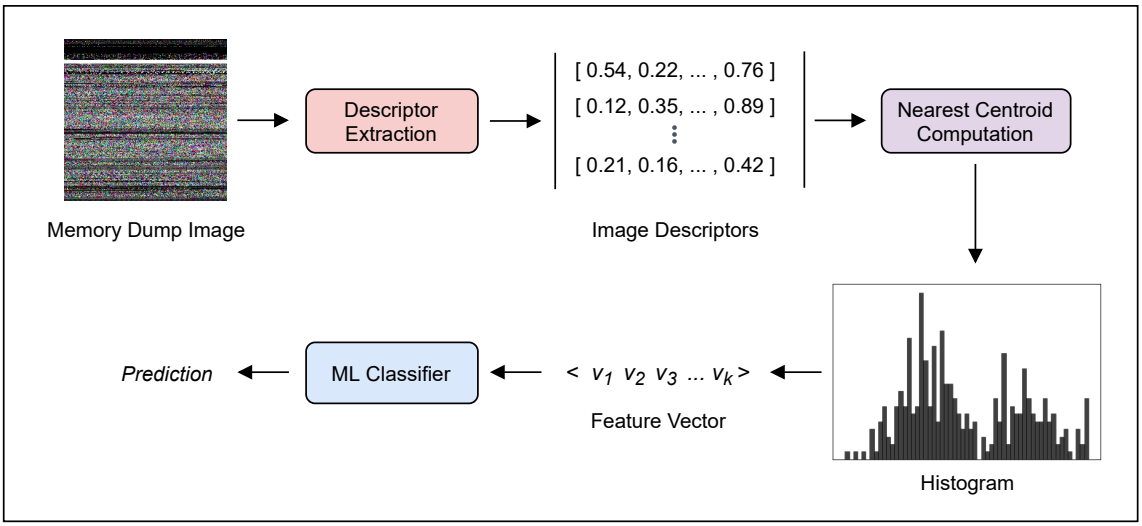


Figure 4.3: Overview of Classification of Memory Dump Images using Descriptors

are quantized using the centroids obtained during the training. The feature vectors are then given as input to the machine learning algorithm, which predicts the class to which the input image belongs. We briefly explain the different machine learning classifiers that are considered.

1. **Decision Tree:** Decision Tree takes into consideration different input attributes to predict the value of the given target attribute. It comprises a set of internal nodes representing a test on an attribute. The test outcomes are represented using branches of the tree, whereas the class labels are represented using the leaf nodes. These are effective as they are fast and provide human-readable rules for classification.

2. **Gaussian Naive Bayes:** Naive Bayes algorithms are a family of supervised learning algorithms that employ the Bayes theorem with an underlying 'naive' assumption that the presence of any particular features is independent of the presence of any other feature for a given class. The Gaussian Naive Bayes algorithm is a special case of Naive Bayes, where it is assumed that the likelihood of the features of the dataset is Gaussian.

3. **k Nearest Neighbour:** k Nearest Neighbour is one of the simplest supervised machine learning algorithms which assigns a label to the given data point p based

on the classes of its k nearest neighbouring data points. k is a positive integer generally much smaller than the number of inputs. Different distance measures like Euclidean or Manhattan are used to compute the nearest neighbours. The advantage of kNN is that there is no training stage. The inference is made on the fly with the labelled training data, and hence it is comparatively easier to add new examples to the training set.

4. **Random Forest:** Random Forest (or Random decision forests) is a type of ensemble learning method and used for predictive modelling. It can be considered as a collection of multiple decision trees trained on a set of randomly chosen samples. The outcome of all decision trees is compiled to bring up the final decision by majority voting. Random decision forests overcome the drawback of decision trees being prone to overfitting.

5. **Support Vector Machines:** Support Vector Machine (SVM) is one of the most widely used supervised machine learning algorithm. It separates data using a hyperplane, which acts as a decision boundary between different classes. The goal of the SVM is to select such a hyperplane that produces the largest margin between the samples of different classes. SVM works not only for linear data but also performs well in case of non-linear data. To handle non-linear data, it uses 'kernels' by transforming data into another dimension that can generate the best possible hyperplane and then performs classification.

We also use a deep learning based classifier, explained in the next subsection, that directly takes the memory dump images for classification instead of relying on the computation of above-mentioned descriptors.

4.3.3 Vision Transformer based Malware Detection

The original Transformer model proposed by Vaswani et al. [85] takes a sequence of words as input and processes it to perform language processing tasks such as translation and sentiment analysis. Dosovitskiy et al. [83] introduced a Vision Transformer

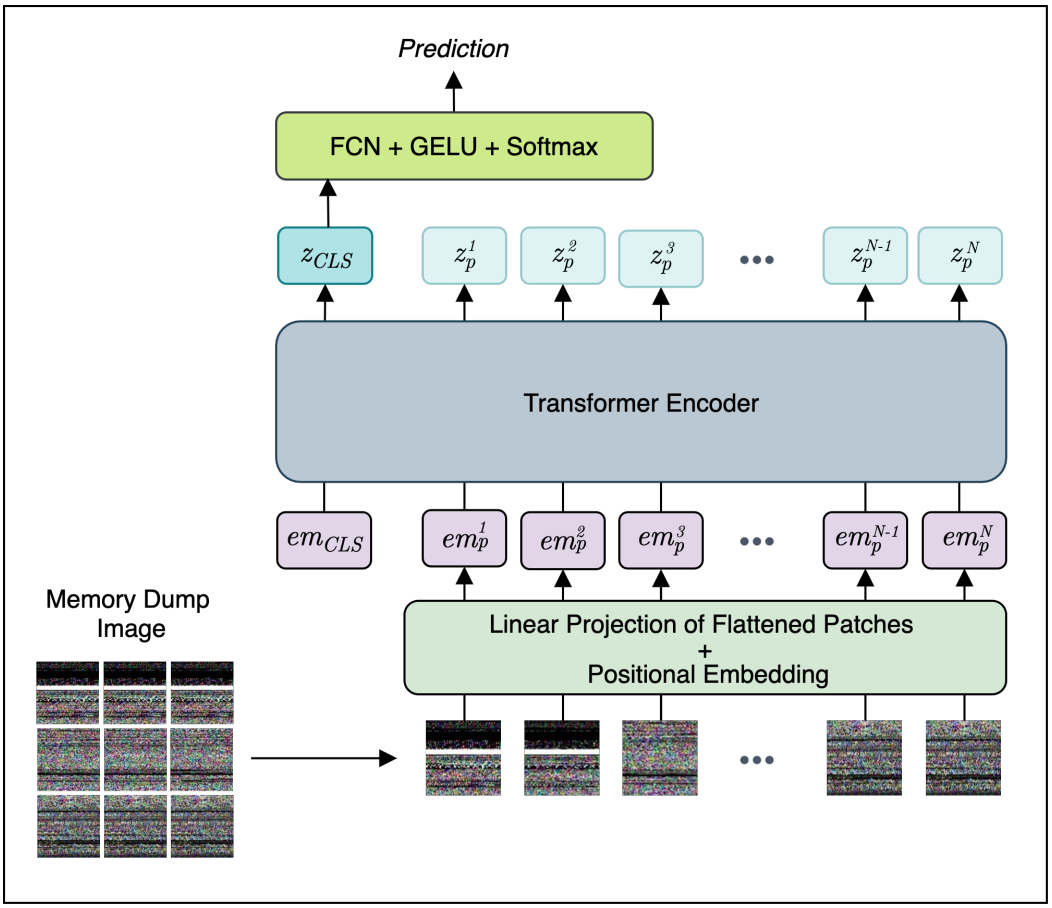


Figure 4.4: ViT Architecture

(ViT), which extends the original Transformer architecture, and obtains state-of-the-art performance on computer vision applications like image classification. We make use of the ViT architecture for the detection and classification of malware using memory dump images as shown in Figure 4.4. The standard Transformer architecture expects a sequence of embeddings, which is a one-dimensional input. Hence to serve an image as an input to the Transformer architecture, we need to reshape the two-dimensional image, $i \in \mathbb{R}^{H \times W \times C}$ to a sequence of flattened two-dimensional square patches, $i_p \in \mathbb{R}^{N \times (P^2 \times C)}$. Here H denotes the height of the image, W denotes the width of the image, C denotes the number of channels, P denotes the height of each square image patch, and $N = (H \times W)/P^2$ denotes the number of patches. The square patches i_p are further projected to patch embeddings $e_p \in \mathbb{R}^{N \times D}$, using a trainable linear projection layer $\mathbf{E} \in \mathbb{R}^{(P^2 \times C^2) \times D}$, as shown in Equation (4.1). Note that D is

the latent vector size, and is constant across all of its layers.

$$e_p^k = i_p^k \cdot \mathbf{E}, \text{ for all } k = 1 \cdots N \tag{4.1}$$

For the purpose of classification, we make use of a special learnable token embedding e_{CLS}. This token is prepended to the start of every sequence of embedding patches.

Transformers are agnostic to the structure and positions of the input elements; hence it requires additional position embeddings to define the position of each patch. If we do not provide this positional information, the input to the model will behave just as a bag of patches [83]. Hence, for encompassing the positional and structural information, learnable position embeddings, $\mathbf{E}_{pos} \in \mathbb{R}^{(N+1)\times D}$ are added to the patch embeddings e_p, to obtain encoded embeddings vectors em_p (as shown in Equation (4.2)).

$$em_{CLS} = e_{CLS} + \mathbf{E}_{pos}^{CLS} \quad \text{and} \quad em_p^k = e_p^k + \mathbf{E}_{pos}^k, \text{ for all } k = 1 \cdots N \tag{4.2}$$

These resultant embeddings vectors em_p and em_{CLS}, are fed as input to the Transformer encoder. In order to harness the contextual information between each patch, the encoder applies self-attention, which captures the relevance amongst every pair of patches. To this end, each embedding em_p is projected to Key K, Query Q and Value V matrices using three corresponding learnable weight matrices. For a given embedding em_p, the self-attention $A \in \mathbb{R}^{N\times D}$ is calculated as product of the *values* to the softmax of the normalized dot product of *keys* and *queries* as shown in Equation (3.3). Further, for capturing multiple complex relationships amongst the patches, the architecture computes multiple attentions for the given patch sequence. These multiple attentions are concatenated together and then passed through a linear layer to get the output z_p. The corresponding output of the e_{CLS} token, z_{CLS} is given as an input to a fully connected layer. The output of this layer is further passed to a softmax layer to obtain a probability distribution vector. The vector denotes the probability for each class. The output for the given input is the class with the highest probability.

4.4 Experiments

In this section, we discuss the dataset used to evaluate our models. Subsequently, we briefly describe the evaluation metrics used to calculate the efficacy of the proposed models. In the next subsection, we discuss the experimental setup and parameter selection, followed by an analysis of extracted features. Finally, we present the results, followed by a performance comparison with previous works.

4.4.1 Dataset

We use a publicly available dataset known as Dumpware10[1] that consists of memory dump images of 4294 portable executables belonging to Windows OS. It has 3686 malware samples belonging to 10 different malware families, whereas the number of benign samples is 608. The dataset is pre-split in training and testing samples with a ratio of 80:20, respectively. The details of the dataset are shown in Table 4.1.

4.4.2 Evaluation Metrics

We use the same set of evaluation metrics described in Section 3.4.2 namely accuracy, precision, recall and F1-score.

4.4.3 Experimental Setup

For our experiments, we use the training and test files which are predetermined as shown in Table 4.1. We briefly discuss the implementation details of the classifiers below:

1. **ML based Classifiers:** We use OpenCV [104] for computing the descriptors. For generating BOVW feature vectors for these descriptors, the k-means clustering algorithm is used in the vocabulary construction step. We use a GPU-enabled *Fast Pytorch Kmeans* library [112], which performs significantly faster than non GPU-enabled implementation. We use the scikit-learn library [113] to

[1]Dataset Link: `https://web.cs.hacettepe.edu.tr/~selman/dumpware10/`

Table 4.1: Characteristics of the Dumpware10 Dataset

Class	**Category**	**Training Samples**	**Testing Samples**	**Total**
Vilsel	Trojan	311	78	389
VBA	Virus	399	100	499
MultiPlug	Adware	390	98	488
InstallCore.C	Adware	376	91	467
BrowseFox	Adware	152	38	190
Dinwod!rfn	Trojan	98	29	127
AutoRun-PU	Worm	158	38	196
Amonetize	Adware	349	87	436
Allaple.A	Worm	349	88	437
Adposhel	Adware	364	93	457
Benign Files	-	487	121	608
	Total	**3433**	**861**	**4294**

implement traditional ML models. The elbow criteria is used to tune the number of neighbours (k) in the kNN algorithm. We observe that for SIFT, ORB and KAZE, the optimal values of k are 3, 7 and 1, respectively. For hypertuning parameters of SVM, we use the grid search cross-validation (GridSearchCV) method. We find that we get optimum results if we choose the kernel as 'RBF', the regularisation parameter (C) as 10 and the kernel coefficient (γ) as 0.001.

2. **Vision Transformer based Classifier:** We use the transformer library of Hugging Face [93] for the implementation and the weights of the ViT model. During the fine-tuning, the model weights are updated using $AdamW$ optimiser [94]. The learning rate *(lr)* is set to 4×10^{-5}. Furthermore, we use cross-entropy loss and the number of epochs as 10 for training the model. The batch size, which is the number of training samples processed in one iteration, is set to 32. For the classification layer, we perform our experiments on two different activation functions, Rectified Linear Units (ReLU) [114], and Gaussian Error

Linear Units (GELU) [115] and find that GELU gives better performance.

4.4.4 Analysing Extracted Features

We perform a qualitative analysis on the different sets of features that are extracted from the memory dump images. In order to do so, we use the t-Distributed Stochastic Neighbour (t-SNE) algorithm [116] to generate a visualisation of the extracted features. The goal of t-SNE is to reduce the dimensionality of high-dimensional data into low-dimensional space such that the points that are close in high-dimensional space remain close in 2-dimensional space.

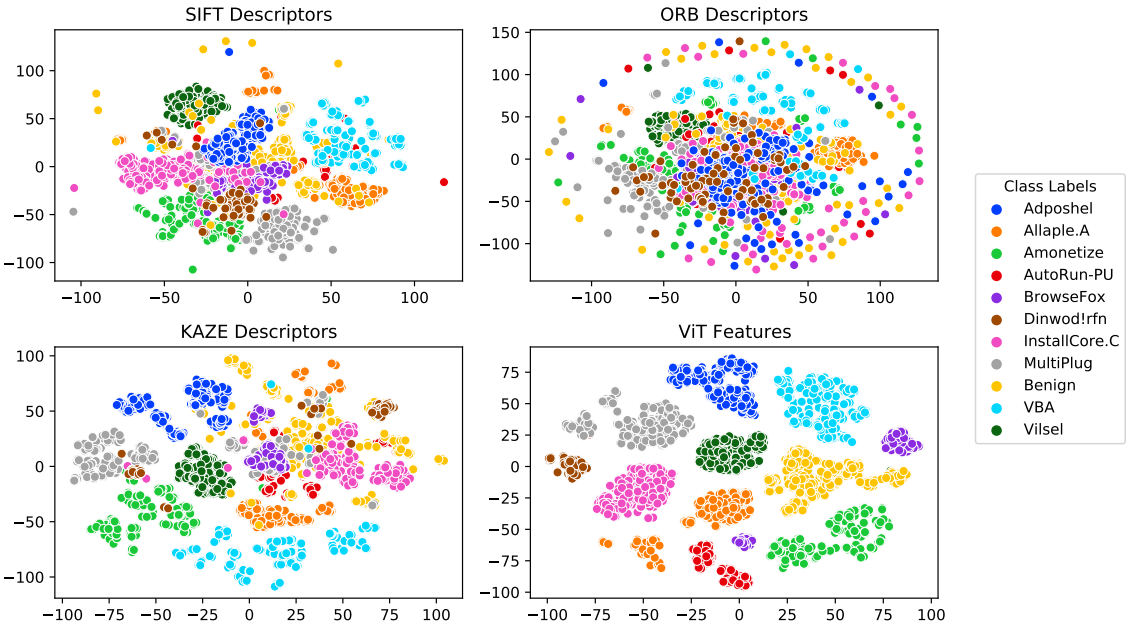


Figure 4.5: Visualisation of Features using t-SNE

Figure 4.5 shows the t-SNE plot for different descriptors like SIFT, ORB, KAZE and also the feature vector corresponding to the $[CLS]$ token of the last layer in ViT. It is important to note that the t-SNE dimensionality reduction technique is unsupervised. Hence it does not use the class labels. The class labels are used only for colouring the points during plotting. We can see that features extracted from the $[CLS]$ token of the ViT have better separable features compared to the features obtained using other techniques. Among the local descriptors, KAZE descriptors give

the best separable features.

4.4.5 Evaluation Results

We perform our experiments with settings mentioned previously for malware detection and classification using the Dumpware10 dataset. The local features like SIFT, ORB and KAZE are extracted, and malware classification performance with different ML algorithms is done. Furthermore, we also compare the performance of such techniques to a deep learning model ViT.

Table 4.2: Comparison of Different Classifiers on Dumpware10 Dataset

Descriptor	**Method**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
SIFT	Decision Tree	0.717	0.723	0.717	0.718
	Gaussian NB	0.723	0.807	0.723	0.732
	kNN	0.823	0.838	0.823	0.807
	Random Forest	0.839	0.854	0.839	0.826
	SVM	0.937	0.936	0.937	0.935
ORB	Decision Tree	0.472	0.494	0.472	0.477
	Gaussian NB	0.708	0.749	0.708	0.709
	kNN	0.681	0.701	0.681	0.681
	Random Forest	0.658	0.752	0.658	0.653
	SVM	0.793	0.801	0.793	0.792
KAZE	Decision Tree	0.817	0.817	0.817	0.816
	Gaussian NB	0.845	0.871	0.845	0.851
	kNN	0.924	0.925	0.924	0.923
	Random Forest	0.932	0.940	0.932	0.932
	SVM	*0.959*	*0.963*	*0.959*	*0.959*
-	ViT	**0.972**	**0.973**	**0.972**	**0.972**

Table 4.2 shows the evaluation based on different metrics. Bold values show the best value for the corresponding metric, whereas values italicised are the second best. It can be observed from the table that SVM performs well for all the local feature

descriptors, and ViT has the best performance overall.

Table 4.3: Confusion Matrix Obtained for KAZE-SVM on Dumpware10 Dataset

Actual \ Predicted	Adposhel	Allaple.A	Amonetize	AutoRun-PU	BrowseFox	Dinwod!rfn.C	InstallCore.C	MultiPlug	Benign	VBA	Vilsel
Adposhel	93	0	0	0	0	0	0	0	0	0	0
Allaple.A	0	85	0	2	1	0	0	0	0	0	0
Amonetize	0	0	87	0	0	0	0	0	0	0	0
AutoRun-PU	0	1	0	31	1	0	0	0	5	0	0
BrowseFox	0	0	0	0	38	0	0	0	0	0	0
Dinwod!rfn	0	0	0	0	0	21	0	0	8	0	0
InstallCore.C	0	0	0	1	0	1	89	0	0	0	0
MultiPlug	0	1	0	2	0	0	0	88	7	0	0
Benign	1	0	0	0	0	0	0	0	120	0	0
VBA	0	0	0	0	0	0	0	0	1	99	0
Vilsel	0	0	0	0	0	0	0	0	3	0	75

Table 4.4: Confusion Matrix Obtained for ViT on Dumpware10 Dataset

Actual \ Predicted	Adposhel	Allaple.A	Amonetize	AutoRun-PU	BrowseFox	Dinwod!rfn.C	InstallCore.C	MultiPlug	Benign	VBA	Vilsel
Adposhel	92	0	0	1	0	0	0	0	0	0	0
Allaple.A	0	86	0	2	0	0	0	0	0	0	0
Amonetize	0	0	86	0	0	0	1	0	0	0	0
AutoRun-PU	0	0	0	36	2	0	0	0	0	0	0
BrowseFox	0	0	0	0	38	0	0	0	0	0	0
Dinwod!rfn	0	0	3	0	0	24	0	0	0	2	0
InstallCore.C	0	0	0	0	1	1	89	0	0	0	0
MultiPlug	0	0	0	0	0	0	0	96	1	0	1
Benign	2	0	0	2	0	1	1	0	115	0	0
VBA	0	0	0	0	0	0	0	0	1	99	0
Vilsel	1	0	0	0	0	0	0	0	1	0	76

We show the class-wise predictions on the test set using confusion matrices in Table 4.3 and Table 4.4 for KAZE-SVM and ViT respectively. We can see that the

Dinwod!rfn malware family is the most difficult to classify as the proportion of misclassification is high in both techniques. A likely reason for this is that the number of samples of this family for training is the least in the dataset. Figure 4.6 shows a plot of Receiver Operating Characteristic (ROC) curves of different classes using different techniques. The best AUC score across all the classes is for ViT, whereas it is the worst for ORB-SVM.

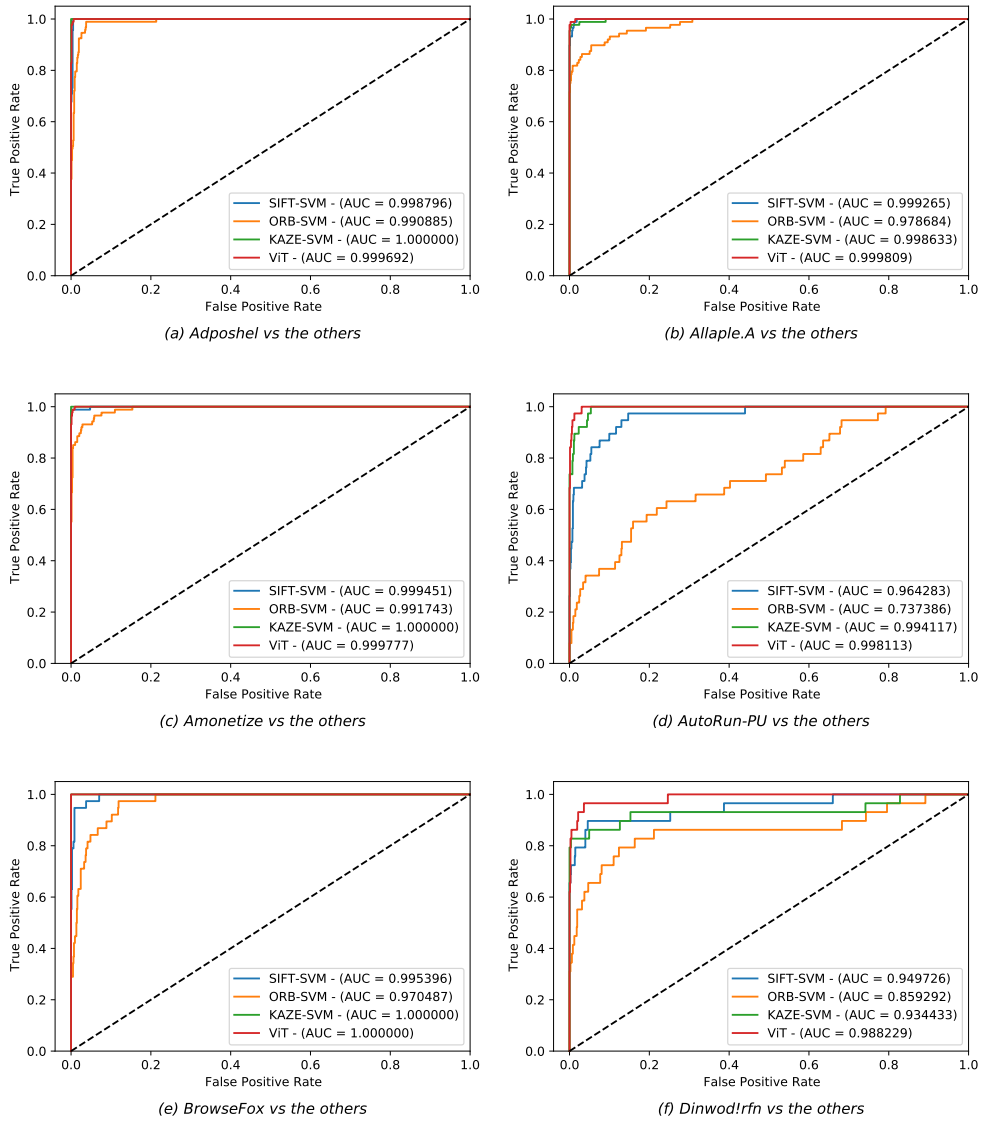
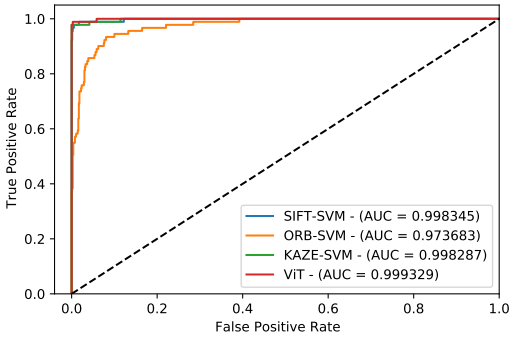
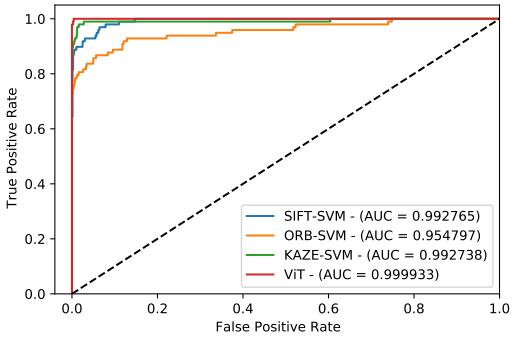


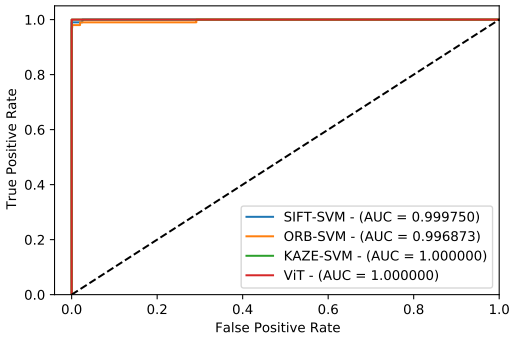
Figure 4.6: ROC Curves of Different Techniques in One vs Rest Setting (continued next page)



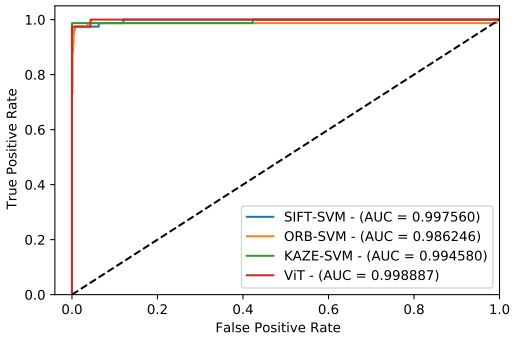
(g) InstallCore.C vs the others



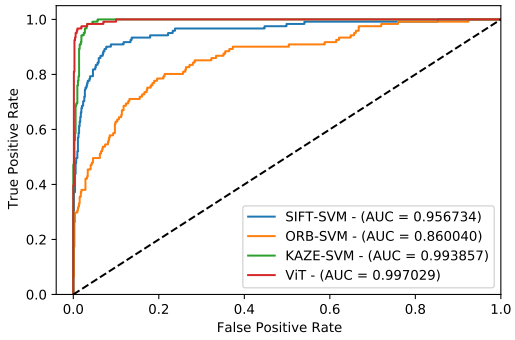
(h) MultiPlug vs the others



(j) VBA vs the others



(k) Vilsel vs the others



(i) Benign vs the others

Figure 4.6: ROC Curves of Different Techniques in One vs Rest Setting

4.4.6 Performance Comparison

We compare the performance of our proposed techniques for malware detection and classification using memory dumps with some other recent works. Nataraj et al. [23] in their work used GIST [31] descriptors that use wavelet decomposition of the images. Features extracted from the images are then used with a k-nearest neighbour classifier for classification. Dai et al. [64] proposed a classification method using the memory dumps. They extract the Histogram of Gradient (HOG) [117] features from the images and use a multilayer perceptron as a classifier. Bozkir et al. [65] used a combination of both GIST and HOG features extracted from the memory dump images. They used the SMO algorithm for training an SVM classifier with an RBF kernel. Rezende et al. [118] used a pre-trained VGG16 model, a deep convolutional neural network, that extracts features from the images. These features are then used with an SVM classifier. The works [23, 118] were originally proposed for detection and classification of malware from images created from portable executable files and not memory dumps. We compare our techniques with theirs on the Dumpware10 dataset as both the previous techniques rely on images to detect and classify the malware.

Table 4.5: Performance Comparison for Malware Detection and Classification on Dumpware10 Dataset

Study	**Technique**	**Accuracy**	**Precision**	**Recall**	**F1-Score**
Nataraj et al. [23]	GIST Descriptor + kNN	0.914	0.915	0.914	0.915
Dai et al. [64]	HOG Descriptor + MLP	0.945	0.946	0.945	0.945
Rezende et al. [118]	VGG16	0.969	0.970	0.969	0.969
Bozkir et al. [65]	GIST + HOG - SVM (RBF SMO)	0.955	0.958	0.951	0.955
Proposed Work	KAZE + SVM	0.959	0.963	0.959	0.959
	ViT	0.972	0.973	0.972	0.972

We compare the performance of the proposed models with the recent works on the Dumpware10 dataset and results are shown in Table 4.5. We can see that the proposed ViT has the best performance in terms of accuracy, precision, recall and F1-score. We can also see that among the techniques that use image descriptors for classification,

the proposed technique using the KAZE descriptor performs better and is comparable to the performance of ViT, which is slightly better.

4.5 Conclusion

Malware are becoming increasingly complex, with many of the recent ones being packed and encrypted. In this chapter, we used a Transformer-based model, ViT, that can detect and classify the malware using the images of the memory dumps. We evaluate this model based on a publicly available dataset, Dumpware10 and show that our technique performs better in comparison to recent other techniques. We also show that traditional machine learning algorithms such as SVM, when used with KAZE descriptors, perform fairly well to detect and classify malware using memory dump images.

Chapter 5

Android Malware Detection

5.1 Introduction

Malware detection and classification methods proposed in Chapter 3 and Chapter 4 are specific to Windows OS. Among all the operating systems available in the market, Windows OS is the most widely used. Closely following it is Android OS, which has become quite popular due to the increased use of mobile devices. Another factor for the growth of Android OS is the flexible policy that does not impose restrictions on the developers on getting their apps published on Google Play Store [119]. The presence of other third-party app markets exacerbates the situation. Owing to these advantages, Android-based devices have become a common target for malware attacks like breach of user privacy, unauthorised banking transactions etc., using malicious applications. In this chapter, we deal with detecting malware that attack Android-based devices.

Our contributions in this chapter are:

- Using static analysis, we extract the opcode sequences present in an Android APK file and use it for detection of Android-based malware.
- We extend the Transformer-based architecture proposed in Chapter 3 for Android malware detection.

5.2 Related Work

As described in Chapter 2, several works [73–75] have used features obtained using static analysis for android malware detection. Some works used a combination of features obtained from both static and dynamic analysis [76, 78]. Deep learning models like RNN, LSTM and CNN were used to handle opcode sequences taken from the entire file for malware detection [78, 80, 81]. Taking inspiration from our previous work in Chapter 3, we use only a few initial opcodes extracted from an android application for malware detection.

5.3 Proposed Work

Unlike the Windows-based Portable Executable files, Android applications have a file format *.apk* i.e. Android Application Package. It is a zip archive that contains all the contents that are necessary to install the app. It contains the following components: i) *AndroidManifest.xml* file which has the application package name, list of permissions that the application requires and other contents; ii) *classes.dex* files, which contain all the methods and classes that are used by the program for execution iii) different *.xml* files that show the layout of the application; iv) other resources such as images, icons and other native libraries. Applications in Android are developed using Java or Kotlin, which are then compiled to an intermediate bytecode format known as Dex (Dalvik Executable). The previously mentioned *classes.dex* file contains the dex instructions. Since this is an unreadable file format, we need to convert the dex file into a human-readable format known as Smali code.

Generally, each instruction in the Smali code contains a single Dalvik opcode followed by multiple operands. An example of a Smali code is shown in Figure 5.1. For the instruction in line 8, the opcode is *new-array*, and the operands are $v4$ and $v5$. Similar to the feature extraction process followed in Section 3.3.3, we consider only the opcodes and discard the associated operands to have a more robust representation against changes made in the operands.

```
        ⋮
8.    new-array v3, v5, [Ljava/lang/Class;
9.    const/16 v4, 0x143
10.   invoke-static {v4}, Lhwmg/vzuskhdfyl/hybzcrkg/a;->a(I)Ljava/lang/String;
11.   move-result-object v4
12.   invoke-static {v4}, Ljava/lang/Class;->forName(Ljava/lang/String;)Ljava/lang/Class;
13.   move-result-object v4
14.   aput-object v4, v3, v7
        ⋮
```

Figure 5.1: A Snippet of Smali Code

Figure 5.2 shows an overall view of the proposed model for android malware detection. For all the APKs marked as benign or malicious in our training set, we initially disassemble them. This is followed by an opcode parser which extracts the opcode mnemonics discarding the operands and outputs the sequence of opcodes corresponding to each application. The extracted opcodes are tokenized and encoded to obtain embedding vectors. The embedding vectors are then fed to the Transformer encoder, which applies the self-attention operation to harness the contextual information between each pair of embedding. In order to capture multiple complex relationships amongst the tokens, the encoder computes multiple attentions for the given sequence of embeddings. This method is explained in detail in Section 3.3.4 for opcodes extracted from Windows executables. Instead of learning the associations between the

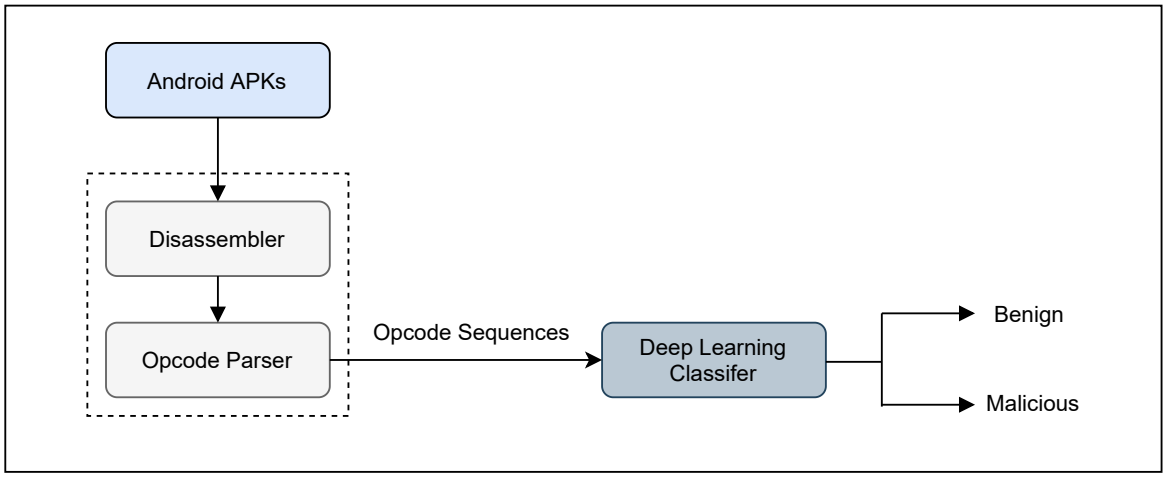


Figure 5.2: An Overview of Proposed Model for Android Malware Detection

opcodes present in Windows-based files, here it learns the association between opcodes present in Android-based files. For testing whether an application is benign or not, we disassemble it and extract the opcode sequence from it. This sequence is passed as an input to our trained Transformer-based classifier, which predicts the probabilities of the file being benign or malicious.

5.4 Experiments

In this section, we describe the dataset used for our experiments. We use the same evaluation metrics as in the previous chapters that are explained in Section 3.4.2. Subsequently, we show the results based on our experiments.

5.4.1 Dataset

For our experiments for android malware detection, we collect benign and malicious applications from AndroZoo [120] which is a repository containing android applications collected from various sources, including Google's Play Store. The collected malware samples belong to different types like Adware, Ransomware, Scareware and SMS malware. The statistics of the dataset is given in Table 5.1. For our experiments we use 2000 samples each of benign and malicious android files.

Table 5.1: Statistics of Dataset for Android Malware Detection

Class	**Type**	**Samples**
Malware	Adware	598
	Ransomware	456
	Scareware	438
	SMS Malware	508
Benign	-	2000
Total		4000

5.4.2 Experimental Setup

We use the previously mentioned dataset for the task of Android malware detection. For disassembling the Android applications in the dataset, we use a Python script that uses an API of Androguard [121] reverse engineering tool. The extracted opcodes from the disassembled files are stored in a CSV file which is later used for training and testing purposes. A train-test split of 70:30 is used, and the opcode sequence length considered is 512. We use the transformer library of Hugging Face [93] for the implementation of the Transformer model. We initially pretrain the model using the masked language modelling task as described in Section 3.3.4. Then it is fine-tuned for the malware detection task. The batch size, which is the number of training samples that are processed in one iteration, is set to 32. The model weights are updated using *AdamW* optimizer [94]. The learning rate *(lr)* is set to 5×10^{-5}. Furthermore, we use cross-entropy loss and the number of epochs as 10 for training the model.

5.5 Results and Discussion

We assess the performance of our proposed technique for android malware detection using opcode sequences. Table 5.2 shows the number of samples predicted correctly and incorrectly for benign and malicious applications in our test set. Unlike the

Table 5.2: Confusion Matrix for Android Malware Detection Experiment

		Predicted: Malware	Predicted: Benign
Actual	Malware	574	35
	Benign	26	565

previous chapters, which included the problem of malware classification, here, we deal with detection only as family labels of the collected dataset could not be obtained.

We also show the Receiver Operating Characteristic (ROC) curves in Figure 5.3 for android malware detection experiment using our proposed model. The AUC value obtained is 0.986813.

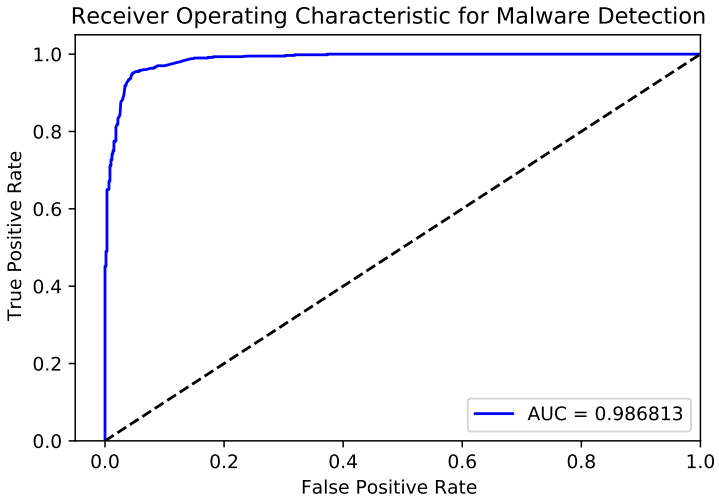


Figure 5.3: ROC Curve for the Android Malware Detection

Table 5.3 shows the performance evaluation of our model on various metrics like accuracy, precision, recall and F1-score. We can see that our model performs fairly well for the detection of android malware.

Table 5.3: Performance of the Proposed Model for Android Malware Detection

Method	**Accuracy**	**Precision**	**Recall**	**F1-score**
Proposed	0.949	0.942	0.956	0.949

We believe that the detection performance of the proposed Transformer-based model can be improved with more training data as Transformer models generally work well for larger datasets [122]. We intend to use different disassemblers to find the one that gives the most accurate transformation from the executable source code to disassembled code. Further, to evaluate the family-wise classification performance of the proposed model, collecting family labelled samples will be looked into.

5.6 Conclusion

Detecting malware in Android-based devices is an important problem due to the widespread used mobile devices. In this chapter, we proposed a Transformer-based model that detects Android malware using short sequence of opcodes. Our experimental evaluation on a dataset that contained four different types of malware showed that the proposed model performs fairly well in detecting Android-based malware.

Chapter 6

Conclusion and Future Work

This chapter summarises the malware detection and classification techniques presented in this thesis and provides future directions for further research in this area. This thesis addresses the problem of malware detection (benign or malware) and also identifying the family of the malware to which it belongs.

We first motivated our thesis by stating that the problem of malware family identification is as important as that of malware detection. We described three different analysis techniques: static, dynamic and hybrid, that are used to detect malware. We also covered the related literature that used the previously mentioned types of analysis for detecting Windows-based malware. We also provided a summary of the works that have been done in the field of Android malware detection. The existing methods that used opcodes for detection had relied on n-grams or the entire opcode sequence present in a file, which are computationally expensive. Using the recent state of the art deep learning model, Transformers, we proposed a method that relied on using only a few opcodes taken from an executable file for malware detection and family identification.

6.1 Thesis Contributions

In this section, we summarise the different techniques proposed in the thesis for malware detection and classification. The first two contributions proposes techniques for the detection and classification of Windows malware, and the third contribution

proposes a technique for Android malware detection. These contributions are summarised in the subsequent three subsections.

6.1.1 Windows Malware Detection and Classification Using Opcode Sequences

Our first contribution of the thesis is a static analysis based technique for detecting and classifying malware executables specific to Windows OS. We disassembled every executable, and the corresponding assembly codes were generated. From the generated assembly codes, we extracted only the opcodes from the assembly level instructions discarding the operands. Corresponding to each executable, a sequence of opcodes was extracted. These extracted opcode sequences are used to train a deep learning model, Transformers, which learns the semantic relationship between opcodes present in a sequence. The conventional sequence-to-sequence architecture like LSTMs used in the literature for opcode sequences takes only short-term context into account. Whereas the Transformer-based model can encapsulate and preserve long dependencies within a sequence. Unlike the previous approaches that used n-grams of opcodes or an entire sequence of opcodes present in a file, the proposed model can detect malware and identify its family using only a short sequence of opcodes extracted from the beginning of a file. Different ablation studies were carried out to understand the impact of tokenizers, opcode sequence length and number of Transformer encoder blocks. We evaluated our model using two different datasets, one containing executables collected by us, whereas the second one is a publicly available Microsoft Malware Family dataset. The comparative results showed that our proposed method outperforms other related works in the literature.

6.1.2 Windows Malware Detection and Classification Using Memory Dumps

Our first proposed work for Windows malware detection and classification summarised in the previous subsection would fall short in two cases: i) when the executable

has an obfuscated code ii) the malware resides in the memory to carry out the attacks, also known as Fileless malware attacks. To deal with this problem, we proposed a dynamic analysis based technique that relies on memory dumps for the detection and classification of Windows-based malware. Since every program, at some point during its execution, ends up on the RAM, analysis of such memory dumps will help us to detect malware. Memory dumps corresponding to each program is extracted from the RAM. Some of the earlier works in the literature relied on the extraction of features like API calls etc., manually from the memory dumps. To avoid the need to rely on expert's knowledge to extract features from the memory dumps, these dumps are converted to images. As the memory dumps are binary files, every three consecutive bytes in the dump file is converted to their unsigned decimal values, which are then RGB encoded to generate pixels of the corresponding image. We followed two different approaches to detect and classify the memory dump images that correspond to benign and malicious samples. In the first approach, we used different image descriptors like SIFT, KAZE and ORB extracted from the memory dump images. These descriptors were used to train different conventional machine learning classifiers for the task of malware detection and classification. In the second approach, we used a Transformer-based model, ViT, which directly takes the memory dump images as inputs for detecting and classifying the malware. We evaluated our proposed techniques using a publicly available dataset, Dumpware10, which consists of dump images of benign samples and malicious samples from 10 different families. We showed that our proposed techniques perform better than other related techniques in the literature.

6.1.3 Android Malware Detection

The previous works summarised above are limited to detecting the windows based malware and their classification into families. However, the problem of malware detection is not restricted to a single operating system. There has been a significant increase in the number of malware attacks on Android-based systems, partly due to the popularity of mobile phones. Hence, we proposed a technique to detect Android-based malware. We extended the previously mentioned technique that used opcode

sequences for Windows malware detection and classification to detect Android-based malware. We used a disassembler to extract the opcodes present in Android applications that have *.apk* (Android Application Package) format. As done previously, from the instructions, we extracted the opcodes discarding the operands. Corresponding to each *apk* file, a sequence of opcodes was generated, which is used to train a similar Transformer-based model that was used earlier. Using only a short sequence of opcodes, the proposed model was able to detect benign and malicious android applications.

6.2 Future Work

Our work on malware detection and family identification can be extended in many ways. Some of the extensions possible are mentioned below.

1. *Use of Hybrid Features:* We had considered opcode sequences as features for the detection and classification of malware. It would be interesting to find out whether using other features such as API calls in combination with the opcodes can improve the performance of the model in the detection and classification of malware.

2. *Concept Drift:* It would be an interesting experiment to find out how frequently the proposed models should be trained with new data to avoid a degradation in the performance. One way to do it can be to use a time-stamped dataset where every sample in the dataset has the time that the sample was created. The model can be trained using the older samples and tested on newer ones.

3. *Security Analysis:* A formal security analysis will be carried out by defining a proper threat model by identifying threat agents that cause harm to an application or computer system. Identifying the vulnerabilities on different levels (application, kernel etc.) of Windows and Android OS that a malware can exploit can be looked into.

4. *Blockchain Technology:* It would be an interesting direction to explore the use of Blockchain technology for the detection and mitigation of malware attacks. One of the main characteristics of Blockchain is decentralisation, which can be used for sharing information of suspected malware rapidly without the intervention of a central organisation. Also, it can be used to trace out the origin of an executable and mitigation the propagation.

Bibliography

[1] Y. Ye, T. Li, S. Zhu, W. Zhuang, E. Tas, U. Gupta, and M. Abdulhayoglu, "Combining File Content and File Relations for Cloud based Malware Detection," in *SIGKDD '11: Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2011, pp. 222–230.

[2] D. Gibert, C. Mateu, and J. Planes, "An End-to-End Deep Learning Architecture for Classification of Malware's Binary Content," in *ICANN '18: Proceedings of Artificial Neural Networks and Machine Learning*. Springer International Publishing, 2018, pp. 383–391.

[3] D. Ucci, L. Aniello, and R. Baldoni, "Survey of Machine Learning techniques for Malware Analysis," *Computers & Security*, vol. 81, pp. 123–147, 2019.

[4] V. S. Sathyanarayan, P. Kohli, and B. Bruhadeshwar, "Signature generation and detection of malware families," in *ACISP '08: Proceedings of the Australasian Conference on Information Security and Privacy*. Springer, 2008, pp. 336–349.

[5] J. Lee, C. Im, and H. Jeong, "A Study of Malware Detection and Classification by Comparing Extracted Strings," in *ICUIMC '11: Proceedings of the International Conference on Ubiquitous Information Management and Communication*. ACM, 2011, p. 75.

[6] C. Varol, A. Varol *et al.*, "Comparison of Pattern Matching Techniques on Identification of Same Family Malware," *International Journal of Information Security Science*, vol. 4, pp. 104–111, 2015.

[7] D. Konopiskỳ, "Malware Detection in Applications based on Presence of Computer Generated Strings," 2018, US Patent App. 15/942,129.

[8] Y. Ye, L. Chen, D. Wang, T. Li, Q. Jiang, and M. Zhao, "SBMDS: An Interpretable String Based Malware Detection System using SVM Ensemble with Bagging," *Journal in Computer Virology*, vol. 5, pp. 283–293, 2009.

[9] R. Lyda and J. Hamrock, "Using Entropy Analysis to Find Encrypted and Packed Malware," *IEEE Security & Privacy*, vol. 5, pp. 40–45, 2007.

[10] I. Sorokin, "Comparing Files using Structural Entropy," *Journal in Computer Virology*, vol. 7, pp. 259–265, 2011.

[11] D. Baysa, R. M. Low, and M. Stamp, "Structural Entropy and Metamorphic Malware," *Journal of Computer Virology and Hacking Techniques*, vol. 9, pp. 179–192, 2013.

[12] T. Abou-Assaleh, N. Cercone, V. Keselj, and R. Sweidan, "N-Gram-Based Detection of New Malicious Code," in *COMPSAC '04: Proceedings of the International Computer Software and Applications Conference*. IEEE Computer Society, 2004, pp. 41–42.

[13] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, and Y. Elovici, "Unknown Malcode Detection via Text Categorization and the Imbalance Problem," in *ISI '08: Proceedings of the IEEE International Conference on Intelligence and Security Informatics*. IEEE, 2008, pp. 156–161.

[14] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas, "Opcode Sequences as Representation of Executables for Data-Mining-Based Unknown Malware Detection," *Information Science*, vol. 231, pp. 64–82, 2013.

[15] Y. Ding and S. Zhu, "Malware Detection Based on Deep Learning Algorithm," *Neural Computing and Applications*, vol. 31, pp. 461–472, 2019.

[16] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin, "MutantX-S: Scalable Malware Clustering Based on Static Features," in *USENIX ATC '13: Proceedings of the USENIX Conference on Annual Technical Conference*. USENIX Association, 2013, pp. 187–198.

[17] B. Cakir and E. Dogdu, "Malware Classification Using Deep Learning Methods," in *ACMSE '18: Proceedings of the ACM South East Conference*. ACM, 2018, pp. 1–10.

[18] Y. Sung, S. Jang, Y.-S. Jeong, and J. H. J. J. Park, "Malware Classification Algorithm Using Advanced Word2Vec-based Bi-LSTM for Ground Control Stations," *Computer Communications*, vol. 153, pp. 342–348, 2020.

[19] S. Jain and Y. K. Meena, "Byte Level n-gram Analysis for Malware Detection," in *ICIP '11: Proceedings of the International Conference on Information Processing*. Springer, 2011, pp. 51–59.

[20] J. Jang, D. Brumley, and S. Venkataraman, "BitShred: Feature Hashing Malware for Scalable Triage and Semantic Analysis," in *CCS '11: Proceedings of the ACM Conference on Computer and Communications Security*. ACM, 2011, pp. 309–320.

[21] E. Raff, J. Sylvester, and C. Nicholas, "Learning the PE Header, Malware Detection with Minimal Domain Knowledge," in *AISec '17: Proceedings of the ACM Workshop on Artificial Intelligence and Security*. ACM, 2017, pp. 121–132.

[22] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. K. Nicholas, "Malware Detection by Eating a Whole EXE," in *AAAI '18: Proceedings of the AAAI Conference on Artificial Intelligence*. Association for the Advancement of Artificial Intelligence, 2018.

[23] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware Images: Visualization and Automatic Classification," in *VizSec '11: Proceedings of the International Symposium on Visualization for Cyber Security*. ACM, 2011, p. 4.

[24] D. Gibert, C. Mateu, J. Planes, and R. Vicens, "Using Convolutional Neural Networks for Classification of Malware Represented as Images," *Journal of Computer Virology and Hacking Techniques*, vol. 15, pp. 15–28, 2019.

[25] E. Rezende, G. Ruppert, T. Carvalho, F. Ramos, and P. De Geus, "Malicious Software Classification using Transfer Learning of ResNet-50 Deep Neural Network," in *ICMLA '17: Proceedings of IEEE International Conference on Machine Learning and Applications (ICMLA)*. IEEE, 2017, pp. 1011–1014.

[26] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *ACM SIGKDD Explorations Newsletter*, vol. 11, pp. 10–18, 2009.

[27] I. Santos, C. Laorden, and P. G. Bringas, "Collective Classification for Unknown Malware detection," in *SECRYPT '11: Proceedings of the International Conference on Security and Cryptography*. IEEE, 2011, pp. 251–256.

[28] I. Santos, F. Brezo, B. Sanz, C. Laorden, and P. G. Bringas, "Using Opcode Sequences in Single-Class Learning to Detect Unknown Malware," *IET Information Security*, vol. 5, pp. 220–227, 2011.

[29] I. Santos, B. Sanz, C. Laorden, F. Brezo, and P. G. Bringas, "Opcode-Sequence-based Semi-supervised Unknown Malware Detection," in *Computational Intelligence in Security for Information Systems*. Springer, 2011, pp. 50–57.

[30] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient Estimation of Word Representations in Vector Space," in *ICLR '13: Proceedings of the International Conference on Learning Representations*, 2013.

[31] A. Oliva and A. Torralba, "Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope," *International Journal of Computer Vision*, vol. 42, pp. 145–175, 2001.

[32] Y. Ye, D. Wang, T. Li, D. Ye, and Q. Jiang, "An Intelligent PE-malware Detection System Based on Association Mining," *Journal in Computer Virology*, vol. 4, pp. 323–334, 2008.

[33] M. N. A. Zabidi, M. A. Maarof, and A. Zainal, "Malware Analysis with Multiple Features," in *ICCMS '12: Proceeesing of the International Conference on Computer Modelling and Simulation.* IEEE, 2012, pp. 231–235.

[34] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware Detection based on Mining API Calls," in *SAC '10: Proceedings of the ACM symposium on applied computing.* ACM, 2010, pp. 1020–1025.

[35] M. Shankarapani, K. Kancherla, S. Ramammoorthy, R. Movva, and S. Mukkamala, "Kernel Machines for Malware Classification and Similarity Analysis," in *IJCNN '10: Proceedings of the International Joint Conference on Neural Networks.* IEEE, 2010, pp. 1–6.

[36] P. Faruki, V. Laxmi, M. S. Gaur, and P. Vinod, "Mining Control Flow Graph as API Call-grams to Detect Portable Executable Malware," in *SIN '12: Proceedings of the International Conference on Security of Information and Networks.* ACM, 2012, pp. 130–137.

[37] J. Kinable and O. Kostakis, "Malware Classification based on Call Graph Clustering," *Journal in Computer Virology*, vol. 7, pp. 233–245, 2011.

[38] "IDA Pro," https://www.hex-rays.com/products/ida, (last accessed on 20-04-2021).

[39] "Radare 2," https://rada.re/n/, (last accessed on 16-01-2021).

[40] M. Hassen and P. K. Chan, "Scalable Function Call Graph-based Malware Classification," in *CODASPY '17: Proceedings of the ACM on Conference on Data and Application Security and Privacy.* ACM, 2017, pp. 239–248.

[41] J. Saxe and K. Berlin, "Deep Neural Network Based Malware Detection using Two Dimensional Binary Program Features," in *MALWARE '15: Proceedings of the International Conference on Malicious and Unwanted Software*. IEEE, 2015, pp. 11–20.

[42] D. Carlin, A. Cowan, P. O'Kane, and S. Sezer, "The Effects of Traditional Anti-Virus Labels on Malware Detection Using Dynamic Runtime Opcodes," *IEEE Access*, vol. 5, pp. 17 742–17 752, 2017.

[43] D. Carlin, P. O'Kane, and S. Sezer, "Dynamic Analysis of Malware using Run-time Opcodes," in *Data Analytics and Decision Support for Cybersecurity*. Springer, 2017, pp. 99–125.

[44] P. O'Kane, S. Sezer, and K. McLaughlin, "Detecting Obfuscated Malware using Reduced Opcode Set and Optimised Runtime Trace," *Security Informatics*, vol. 5, p. 2, 2016.

[45] K. Pearson, "LIII. On Lines and Planes of Closest Fit to Systems of Points in Space," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 2, pp. 559–572, 1901.

[46] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane, "Graph-based Malware Detection using Dynamic Analysis," *Journal in Computer Virology*, vol. 7, pp. 247–258, 2011.

[47] C. Storlie, B. Anderson, S. V. Wiel, D. Quist, C. Hash, and N. Brown, "Stochastic Identification of Malware with Dynamic Traces," *The Annals of Applied Statistics*, pp. 1–18, 2014.

[48] N. Kheir, "Behavioral Classification and Detection of Malware through HTTP User Agent Anomalies," *Journal of Information Security and Applications*, vol. 18, pp. 2–13, 2013.

[49] D. Bekerman, B. Shapira, L. Rokach, and A. Bar, "Unknown Malware Detection using Network Traffic Classification," in *CNS '15: Proceedings of the IEEE*

Conference on Communications and Network Security. IEEE, 2015, pp. 134–142.

[50] G. Zhao, K. Xu, L. Xu, and B. Wu, "Detecting APT Malware Infections Based on Malicious DNS and Traffic Analysis," *IEEE Access*, vol. 3, pp. 1132–1142, 2015.

[51] A. Boukhtouta, S. A. Mokhov, N.-E. Lakhdari, M. Debbabi, and J. Paquet, "Network Malware Classification Comparison using DPI and Flow Packet Headers," *Journal of Computer Virology and Hacking Techniques*, vol. 12, pp. 69–100, 2016.

[52] P. Prasse, L. Machlica, T. Pevný, J. Havelka, and T. Scheffer, "Malware Detection by Analysing Encrypted Network Traffic with Neural Networks," in *ECML PKDD '17: Proceedings of the European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Springer, 2017, pp. 73–88.

[53] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic Analysis of Malware Behavior using Machine Learning," *Journal of Computer Security*, vol. 19, pp. 639–668, 2011.

[54] D. Uppal, R. Sinha, V. Mehra, and V. Jain, "Malware Detection and Classification Based on Extraction of API Sequences," in *ICACCI '14: Proceedings of the International conference on advances in computing, communications and informatics*. IEEE, 2014, pp. 2337–2342.

[55] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale Malware Classification using Random Projections and Neural Networks," in *ICASSP '13: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2013, pp. 3422–3426.

[56] W. Huang and J. W. Stokes, "MtNet: A Multi-task Neural Network for Dynamic Malware Classification," in *DIMVA '16: Proceedings of the International*

conference on detection of intrusions and malware, and vulnerability assessment. Springer, 2016, pp. 399–418.

[57] H. S. Galal, Y. B. Mahdy, and M. A. Atiea, "Behavior-based Features Model for Malware Detection," *Journal of Computer Virology and Hacking Techniques*, vol. 12, pp. 59–67, 2016.

[58] B. Athiwaratkun and J. W. Stokes, "Malware Classification with LSTM and GRU Language Models and a Character-level CNN," in *ICASSP '17: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing.* IEEE, 2017, pp. 2482–2486.

[59] B. Kolosnjaji, A. Zarras, G. D. Webster, and C. Eckert, "Deep Learning for Classification of Malware System Call Sequences," in *AI '16: Proceedings of the Advances in Artificial Intelligence.* Springer, 2016, pp. 137–149.

[60] T. Teller and A. Hayon, "Enhancing Automated Malware Analysis Machines with Memory Analysis," *Black Hat USA*, 2014.

[61] R. Mosli, R. Li, B. Yuan, and Y. Pan, "Automated Malware Detection using Artifacts in Forensic Memory Images," in *HST '16: Proceedings of IEEE Symposium on Technologies for Homeland Security.* IEEE, 2016, pp. 1–6.

[62] A. Case and G. G. Richard III, "Detecting Objective-C Malware through Memory Forensics," *Digital Investigation*, vol. 18, pp. S3–S10, 2016.

[63] D. Javaheri and M. Hosseinzadeh, "A Framework for Recognition and Confronting of Obfuscated Malwares based on Memory Dumping and Filter Drivers," *Wireless Personal Communications*, vol. 98, pp. 119–137, 2018.

[64] Y. Dai, H. Li, Y. Qian, and X. Lu, "A Malware Classification Method based on Memory Dump Grayscale Image," *Digital Investigation*, vol. 27, pp. 30–37, 2018.

[65] A. S. Bozkir, E. Tahillioglu, M. Aydos, and I. Kara, "Catch them Alive: A Malware Detection Approach through Memory Forensics, Manifold Learning and Computer Vision," *Computers & Security*, vol. 103, p. 102166, 2021.

[66] A. Pektaş and T. Acarman, "Classification of Malware Families Based on Runtime Behaviors," *Journal of Information Security and Applications*, vol. 37, pp. 91–100, 2017.

[67] R. Islam, R. Tian, L. M. Batten, and S. Versteeg, "Classification of Malware Based on Integrated Static and Dynamic Features," *Journal of Network and Computer Applications*, vol. 36, pp. 646–656, 2013.

[68] W. Han, J. Xue, Y. Wang, Z. Liu, and Z. Kong, "MalInsight: A Systematic Profiling Based Malware Detection Framework," *Journal of Network and Computer Applications*, vol. 125, pp. 236–250, 2019.

[69] W. Han, J. Xue, Y. Wang, L. Huang, Z. Kong, and L. Mao, "MalDAE: Detecting and Explaining Malware Based on Correlation and Fusion of Static and Dynamic Characteristics," *Computers & Security*, vol. 83, pp. 208–233, 2019.

[70] B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, "Empowering Convolutional Networks for Malware Classification and Analysis," in *IJCNN '17: Proceedings of the International Joint Conference on Neural Networks*. IEEE, 2017, pp. 3838–3845.

[71] N. Kumar, S. Mukhopadhyay, M. Gupta, A. Handa, and S. K. Shukla, "Malware classification Using Early Stage Behavioral Analysis," in *AsiaJCIS '19: Proceedings of the Asia Joint Conference on Information Security*. IEEE, 2019, pp. 16–23.

[72] M. Rhode, L. Tuson, P. Burnap, and K. Jones, "LAB to SOC: Robust Features for Dynamic Malware Detection," in *DSN '19: Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks–Industry Track*. IEEE, 2019, pp. 13–16.

[73] R. Vinayakumar, K. Soman, and P. Poornachandran, "Deep Android Malware Detection and Classification," in *ICACCI '17: Proceedings of the International Conference on Advances in Computing, Communications and Informatics*. IEEE, 2017, pp. 1677–1683.

[74] R. Nix and J. Zhang, "Classification of Android apps and Malware using Deep Neural Networks," in *IJCNN'17: Proceedings of the International joint conference on neural networks*. IEEE, 2017, pp. 1871–1878.

[75] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "Androdialysis: Analysis of Android Intent Effectiveness in Malware Detection," *Computers & Security*, vol. 65, pp. 121–134, 2017.

[76] L. Taheri, A. F. A. Kadir, and A. H. Lashkari, "Extensible Android Malware Detection and Family Classification using Network-flows and API-calls," in *ICCST '19: Proceedings of the International Carnahan Conference on Security Technology*. IEEE, 2019, pp. 1–8.

[77] M. Al-Fawa'reh, A. Saif, M. T. Jafar, and A. Elhassan, "Malware Detection by Eating a Whole APK," in *ICITST '20: Proceedings of the International Conference for Internet Technology and Secured Transactions*. IEEE, 2020, pp. 1–7.

[78] M. Almahmoud, D. Alzu'bi, and Q. Yaseen, "ReDroidDet: Android Malware Detection Based on Recurrent Neural Network," *Procedia Computer Science*, vol. 184, pp. 841–846, 2021.

[79] "Using opcode sequences to detect malicious android applications."

[80] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé *et al.*, "Deep Android Malware Detection," in *CODASPY '17: Proceedings of the seventh ACM on conference on data and application security and privacy*. ACM, 2017, pp. 301–308.

[81] R. Mateless, D. Rejabek, O. Margalit, and R. Moskovitch, "Decompiled APK based Malicious Code Classification," *Future Generation Computer Systems*, vol. 110, pp. 135–147, 2020.

[82] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," in *NAACL-HLT '19: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. ACL, 2019, pp. 4171–4186.

[83] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, "An Image is Worth 16x16 words: Transformers for Image Recognition at Scale," in *ICLR '21: Proceedings of the International Conference on Learning Representations*. OpenReview.net, 2021, pp. 60–90.

[84] A. Shabtai, R. Moskovitch, C. Feher, S. Dolev, and Y. Elovici, "Detecting Unknown Malicious Code by Applying Classification Techniques on OpCode Patterns," *Security Informatics*, vol. 1, pp. 1–22, 2012.

[85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is All you Need," in *NIPS '17: Proceedings of the Advances in Neural Information Processing Systems*. Curran Associates Inc., 2017, pp. 5998–6008.

[86] S. Khan, M. Naseer, M. Hayat, S. W. Zamir, F. S. Khan, and M. Shah, "Transformers in Vision: A Survey," *arXiv preprint arXiv:2101.01169*, vol. abs/2101.01169, 2021.

[87] A. M. P. Braşoveanu and R. Andonie, "Visualizing Transformers for NLP: A Brief Survey," in *IV'20: Proceedings of the International Conference on Information Visualisation*. IEEE, 2020, pp. 270–279.

[88] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBerta: A Robustly Optimized BERT Pretraining Approach," *arXiv preprint arXiv:1907.11692*, 2019.

[89] "VirusTotal," 2021. [Online]. Available: https://www.virustotal.com

[90] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft Malware Classification Challenge," *arXiv preprint arXiv:1802.10135*, vol. abs/1802.10135, 2018.

[91] "Microsoft Malware Classification Challenge (BIG 2015)," 2015. [Online]. Available: https://www.kaggle.com/c/malware-classification

[92] "Ghidra," https://ghidra-sre.org, (last accessed on 12-11-2020).

[93] T. Wolf, J. Chaumond, L. Debut, V. Sanh, C. Delangue, A. Moi, P. Cistac, M. Funtowicz, J. Davison, S. Shleifer *et al.*, "Transformers: State-of-the-art Natural Language Processing," in *EMNLP '20: Proceedings of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. ACL, 2020, pp. 38–45.

[94] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," in *ICLR '19: Proceedings of the International Conference on Learning Representations*, 2019.

[95] M. Schuster and K. Nakajima, "Japanese and Korean Voice Search," in *ICASSP '12: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 2012, pp. 5149–5152.

[96] S. Gao, M. Alawad, M. T. Young, J. Gounley, N. Schaefferkoetter, H.-J. Yoon, X.-C. Wu, E. B. Durbin, J. Doherty, A. Stroup *et al.*, "Limitations of Transformers on Clinical Text Classification," *IEEE Journal of Biomedical and Health Informatics*, 2021.

[97] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "DistilBERT, a Distilled Version of BERT: Smaller, Faster, Cheaper and Lighter," *arXiv preprint arXiv:1910.01108*, vol. abs/1910.01108, 2019.

[98] S. Si, R. Wang, J. Wosik, H. Zhang, D. Dov, G. Wang, and L. Carin, "Students Need More Attention: BERT-based Attention Model for Small Data with Application to Automatic Patient Message Triage," in *MLHC '20: Proceedings of the Machine Learning for Healthcare Conference*. PMLR, 2020, pp. 436–456.

[99] Y. Li, S. Rao, J. R. A. Solares, A. Hassaine, R. Ramakrishnan, D. Canoy, Y. Zhu, K. Rahimi, and G. Salimi-Khorshidi, "BEHRT: Transformer for Electronic Health Records," *Scientific Reports*, vol. 10, pp. 1–12, 2020.

[100] J. H. Friedman, "Greedy Function Approximation: A Gradient Boosting Machine," *Annals of Statistics*, vol. 29, pp. 1189–1232, 2001.

[101] S. Kumar *et al.*, "An Emerging Threat Fileless Malware: A Survey and Research Challenges," *Cybersecurity*, vol. 3, pp. 1–12, 2020.

[102] B. Sanjay, D. Rakshith, R. Akash, and V. V. Hegde, "An Approach to Detect Fileless Malware and Defend its Evasive Mechanisms," in *CSITSS '18: Proceedings of the International Conference on Computational Systems and Information Technology for Sustainable Solutions*. IEEE, 2018, pp. 234–239.

[103] "ProcDump," https://docs.microsoft.com/en-us/sysinternals/downloads/procdump, (last accessed on 21-04-2021).

[104] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, vol. 25, pp. 120–125, 2000.

[105] M. Aly, P. Welinder, M. Munich, and P. Perona, "Automatic discovery of image families: Global vs. local features," in *ICIP '09:Proceedings of the IEEE International Conference on Image Processing*. IEEE, 2009, pp. 777–780.

[106] D. G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, vol. 60, pp. 91–110, 2004.

[107] E. Rublee, V. Rabaud, K. Konolige, and G. Bradski, "ORB: An Efficient Alternative to SIFT or SURF," in *ICCV '11: Proceedings of the International Conference on Computer Vision*. IEEE, 2011, pp. 2564–2571.

[108] E. Rosten and T. Drummond, "Machine Learning for High-Speed Corner Detection," in *ECCV '06: Proceedings of the European Conference on Computer Vision*. Springer, 2006, pp. 430–443.

[109] M. Calonder, V. Lepetit, C. Strecha, and P. Fua, "Brief: Binary Robust Independent Elementary Features," in *ECCV '10: Proceedings of the European conference on computer vision*. Springer, 2010, pp. 778–792.

[110] L. Yi-bo and L. Jun-Jun, "Harris Corner Detection Algorithm based on Improved Contourlet Transform," *Procedia Engineering*, vol. 15, pp. 2239–2243, 2011.

[111] P. F. Alcantarilla, A. Bartoli, and A. J. Davison, "KAZE Features," in *ECCV '12: Proceedings of the European conference on computer vision*. Springer, 2012, pp. 214–227.

[112] DeMoriarty, "Fast Pytorch Kmeans," https://github.com/DeMoriarty/fast_pytorch_kmeans, 2020.

[113] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[114] V. Nair and G. E. Hinton, "Rectified Linear Units Improve Restricted Boltzmann Machines," in *ICML '10: Proceeedings of the International Conference on Machine Learning*. Omnipress, 2010, pp. 807–814.

[115] D. Hendrycks and K. Gimpel, "Gaussian Error Linear Units (GELUs)," *arXiv preprint arXiv:1606.08415*, 2016.

[116] L. Van der Maaten and G. Hinton, "Visualizing Data using t-SNE," *Journal of Machine Learning Research*, vol. 9, pp. 2579–2605, 2008.

[117] C. Shu, X. Ding, and C. Fang, "Histogram of the Oriented Gradient for Face Recognition," *Tsinghua Science and Technology*, vol. 16, pp. 216–224, 2011.

[118] E. Rezende, G. Ruppert, T. Carvalho, A. Theophilo, F. Ramos, and P. de Geus, "Malicious Software Classification using VGG16 Deep Neural Network's Bottleneck Features," in *ITNG '18: Proceedings of the Information Technology-New Generations*. Springer, 2018, pp. 51–59.

[119] S. Farooqi, Á. Feal, T. Lauinger, D. McCoy, Z. Shafiq, and N. Vallina-Rodriguez, "Understanding Incentivized Mobile App Installs On Google Play Store," in *IMC '20: Proceedings of the ACM Internet Measurement Conference*. ACM, 2020, pp. 696–709.

[120] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "AndroZoo: Collecting Millions of Android Apps for the Research Community," in *MSR '16: Proceedings of the International Conference on Mining Software Repositories*. IEEE, 2016, pp. 468–471.

[121] A. Desnos and G. Gueguen, "Androguard : Reverse Engineering, Malware and Goodware Analysis of Android Applications," 2013, https://github.com/androguard/androguard, (last accessed at 30-06-2021.

[122] A. Ezen-Can, "A Comparison of LSTM and BERT for Small Corpus," *arXiv preprint arXiv:2009.05451*, vol. abs/2009.05451, 2020.

Publications

1. **Fyse Nassar**, Neminath Hubballi, "Malware Detection and Family Identification with BERT", Journal of Computer Virology and Hacking Techniques, Springer (Submitted)