UNIVERSITY OF MACEDONIA

DEPARTMENT OF APPLIED INFORMATICS

M.Sc IN APPLIED INFORMATICS

XLCNN: PRE-TRAINED TRANSFORMER MODEL FOR MALWARE DETECTION

M.Sc Thesis

of

Konstantinos Giapantzis

Thessaloniki, 22/02/2022

XLCNN: PRE-TRAINED TRANSFORMER MODEL FOR MALWARE DETECTION

Konstantinos Giapantzis

Diploma Degree in Materials & Science Engineering,
University of Ioannina (2018)

M.Sc Thesis

submitted for the partial fulfillment of its requirements for

M.Sc IN APPLIED INFORMATICS

Mavridis Ioannis
Chalkidis Spyridon

It was approved by the three-member examination committee on 22/02/2022

Mavridis Ioannis	Chalkidis Spyridon	Refanidis Ioannis
....	....	....

Konstantinos Giapantzis

Abstract

The present thesis describes a Transformer-based neural network model that was developed in order to detect malicious software. We believe that the scientific community should take advantage of the contribution of Transformer models in the field of cybersecurity and go beyond the limits set by the classic natural language processing. For this purpose a new and more sophisticated algorithm was created based on the methodology used by the XLNet neural network which was proposed by the Google AI Brain Team. The proposed XLCNN model detects malicious code with a higher success rate than its predecessor. The method of detecting malware is based on the extraction and analysis of metadata contained in Windows executable files. From the experiments carried out, it was found that the size and architecture of the feed-forward neural network in combination with the size of its input is one of the most important factors of XLCNN for classification problems. To justify proving the concept of XLCNN as an effective approach to detecting malware, the success rate of the algorithm was measured for a finite number of epochs compared to XLNet using exactly the same parameters as the same inputs. Using this network has proven to be not only a reliable way for security researchers to detect malware, but also an effective and highly accurate method that offers high accuracy of 95.07%.

Keywords: neural network, Transformers, XLNet, XLCNN, malware detection, metadata

Acknowledgements

I would like to thank Dr. Ioannis Mavridis, Professor of Information Systems Security, for the confidence he has shown in my ability to carry out this Master thesis. Also, I want to thank Dr. Spyros T. Halkidis, for the specialized knowledge he provided me with on the part of artificial intelligence, but also his guidance throughout the research.

Table of contents

1 Introduction....1
1.1 Statement....1
1.2 Motivation....2
1.3 Purpose - Objectives....3
1.4 Contribution....3
1.5 Related Work....3
1.6 Structure of the study....4
2 Background....5
2.1 Malware types....5
2.1.1 Virus....5
2.1.2 Trojans....5
2.1.3 Worms....7
2.1.4 Adware....8
2.1.5 Rootkit....9
2.1.6 Bots....9
2.1.7 Ransomware....9
2.2 Malware analysis....11
2.2.1 Static Analysis....12
2.2.1.1 Shortcomings of Static Analysis....15
2.2.1.2 Signature based Analysis....15
2.2.2 Dynamic Analysis....17
2.2.2.1 Function Call Monitoring....19
2.2.2.2 Function hooking implementation....22
2.2.2.3 Information Flow Tracking....23
2.3 Malware Detection....24
2.3.1 Windows PE....24
2.3.2 Obfuscation Techniques....26
2.3.2.1 Packed Malware....26
2.3.2.2 Oligomorphic Malware....26
2.3.2.3 Polymorphic Malware....26
2.3.2.4 Metamorphic Malware....27

3 Machine Learning....28
3.1 CNN....28
3.2 RNN....29
3.3 Transformers....30
3.3.1 Attention Mechanism....31
3.3.2 The Transformer model architecture....31
3.3.3 Scaled Dot-Product Attention....33
3.3.4 Multi-Head Attention....34
3.3.5 Embeddings and Softmax....35
4 Transformer Models....36
4.1 XLNet....36
4.1.1 Characteristics....37
4.1.2 Architecture....38
4.1.3 Two-Stream Self-Attention....39
4.1.4 Partial Prediction....40
4.1.5 Relative Segment Encodings....41
4.1.6 Comparative analysis....41
4.2 Other Transformer Models....43
5 The proposed malware detection approach....45
5.1 Backround Architecture....45
5.1.1 Sequence-to-Sequence....45
5.1.2 AdamW Optimizer....46
5.1.3 Linear Schedule with Warmup....47
5.1.4 Embeddings....48
5.1.5 Dropout....48
5.1.6 Activation function....49
5.2 The Proposed XLCNN Model....51
5.3 Application on malware detection....56
5.3.1 Source collection for malware detection....57
5.3.2 Metadata extraction....57
5.4 Experimental implementation and results....58
5.4.1 Comparison with XLNet....61
6 Epilogue....63

6.1 Summary and conclusions..63
6.2 Limits and limitations of research..64
6.3 Future Extensions..64

Image catalog

Figure 1. Total number of malware detected by year (in million) [75]..............................6
Figure 2. Screenshot of the ransom note left on an infected system by WannaCry [83]. .10
Figure 3: Signature-based procedure [42]..16
Figure 4: Dynamic malware analysis steps [46]..18
Figure 5. Information Flow Tracking [54]..24
Figure 6: Figure 1. General C/C++ compiler PE file structure [57].................................25
Figure 7: Local connections in the architecture of the CNN [63].....................................28
Figure 8: Recurrent Neural Network (RNNs) Architecture...30
Figure 9. The Transformer - model architecture..32
Figure 10. (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of severalattention layers running in parallel..33
Figure 11. XLNet Architecture [6]..38
Figure 12: Feed forward convolutional neural network architecture with input matrix [512 x 1 x 512] and output matrix [512 x 1 x 2048]...52
Figure 13: XLCNN feed forward neural network structure...54
Figure 14: Process of classifying executable files into malicious and benign..................56
Figure 15: Confusion matrix for test set...59
Figure 16: XLCNN training process...60
Figure 17: XLCNN testing process...60
Figure 18: XLCNN training process per iteration..61

Table catalog

Table 1. Static analysis results for malware detection..14
Table 2. Fair comparison with BERT. All models are trained using the same data and hyperparameters as in BERT. We use the best of 3 BERT variants for comparison; i.e., the original BERT, BERT with whole word masking, and BERT without next sentence prediction...42
Table 3. Comparison with state-of-the-art results on the test set of RACE, a reading comprehension task, and on ClueWeb09-B, a document ranking task. * indicates using ensembles. indicates our implementations. “Middle” and “High” in RACE are two subsets representing middle and high school difficulty levels. All BERT, RoBERTa, and XLNet results are obtained with a 24-layer architecture with similar model sizes (BERT-Large)..42
Table 4. Comparison with state-of-the-art error rates on the test sets of several text classification datasets. All BERT and XLNet results are obtained with a 24-layer architecture with similar model sizes (BERT-Large)...43
Table 5. Results on GLUE. * indicates using ensembles, and † denotes single-task results in a multi-task row. All dev results are the median of 10 runs. The upper section shows direct comparison on dev data and the lower section shows comparison with state-of-the-art results on the public leaderboard...43
Table 6. The results for sensitivity and specificity for XLCNN transformer model.........59
Table 7. Comparison with state-of-the-art results on the test set between XLCNN and XLNet..62

Abbreviations

IFT - Information Flow Tracking

CFG - Control Flow Graph

LBP - Local Binary Pattern

UPX - Ultimate Packer for Executables

ASP - Accelerated Security Path

API - Application Programming Interface

SSDT - SQL Server Data Tools

IDT - Interrupt Descriptor Table

DKOM - Dani Icon Manipulation

GPU - Graphics Processing Unit

CPU - Gentral Processing Unit

PSI - Printable Strings Information

Adware - Advertising Supported Software

C&C – Command and Control

SVM – Support Vector Machine

NLP - Natural Language Processing

RNN - Recurrent Neural Network

LSTM - Long Short Term Memory

GRU - Gated Recurrent Units

CNN – Convolutional Neural Networks

AR – Autoregressive

ESEC - European Software Engineering Conference

FSE - Foundations of Software Engineering

GELUs - Gaussian Error Linear Units

ReLUs - Rectified Linear Units

PE – Portable Executable

FPR – False Positive Rate

TNR – True Negative Rate

PPV - Positive Predictive Value

NPV - Negative Predictive Value

FDR - False Fiscovery Rate

ACC - Accuracy

1 Introduction

1.1 Statement

Malware is a program developed with the intent of breaking into a system to monitor, intercept personal information, encrypt data or require ransom [125]. Due to the critical threat posed by malware in cyberspace, many different methods and analysis tools have been developed to detect it. One of these methods computes the file of the under condition signature and compares it to the ones stored in a database containing signatures of known malware. The main disadvantage of this method, which most antivirus tools use, is that, if the code is modified through processing called obfuscation, then it is impossible to detect the existence of malware. Another disadvantage of this method is that, if a new malware with unknown signature is used, then it is practically impossible to detect it.

Another detection technique is based on malware behavior. The detection process is particularly tedious as it requires isolating the file and placing it in a secure environment, such as sandbox, and then supervising its behaviour by qualified personnel. Observing behavior requires a lot of dedication and time while at the same time it may not be effective as concealment techniques are improved and malware evolves to avoid being detected. Therefore, more complex, mathematically intelligent and automated methods are required such as machine learning. Machine learning techniques, especially Deep Learning is an excellent technique that deals with data variants [65] because not only can it learn the given feature during the training process, but also automatically extracts features from data to achieve the goal of classification [1]. When an infected file is given as input to a deep learning algorithm, according to the characteristics it has learned from the training process, it can identify whether it is either malicious or benign.

The problem of detecting malware is considered a classification problem that has two categories. One category is benign software and the other is malicious. The malware and the benign category are then treated as natural language processing (NLP) models, as the metadata of each file can be considered as sequence of words which can be used as input to the neural network. Metadata are a set of ASCII characters, many of which are impossible to decode by the human agent even with the help of specialized programs, let alone capable of extracting those features that contribute to its grouping. For this reason we decided to evaluate some deep learning algorithms such as RNN [2], LSTM [3] and

GRU [4] which have been firmly established as state-of-the-art approaches to sequence modeling and transfer problems such as language modeling and automatic translation.

After research it was found that there is a set of algorithms that is more accurate and less time consuming than all the above methods. Initially, algorithms using attention mechanisms [5] was introduced to solve machine translation problems. Transformer models gradually replaced RNNs in mainstream **N**atural Language Processing NLP. The Transformer model architecture takes a new approach to machine learning as it completely eliminates repetition. Transformers create attributes of each word using an attention mechanism to understand how important all the other words in the sentence are. Knowing these the renewed features of the word are simply the sum of the number of linear transformations of all the words' features weighted by the average number of linear transformations [5].

Despite the great effectiveness of the XLNet [6] Transformer model, to the best of our knowledge, no other security research has implemented malware detection techniques for Windows executables files using the XLNet model. In the present research an XLNet model for the classification problem was implemented and then a completely new Transformer model named XLCNN was proposed and developed. After comparing these two algorithms, it turned out that the latter is more accurate in detecting new malware.

XLCNN uses a complex architecture in the feed forward neural network and this is the main feature that distinguishes it from other Transformer models. The contribution to the scientific community, both at the theoretical level as a new model was proposed and at the level of implementation, is the provision of a pre-trained Transformer model which will identify new malware using their metadata.

1.2 Motivation

The decision to propose the XLCNN came from the combination of cyber security with Transformer models in an effort to take advantage of the possibilities offered by artificial intelligence. The main pillar for the creation of XLCNN was the research effort of the Google team that created the XLNet model. XLCNN exceeded the capabilities of its predecessor as a different feedforward neural network was implemented.

1.3 Purpose - Objectives

The purpose of this thesis is to create a state of the art deep learning model, which is able to classify executable files as malicious and benign. This goal was achieved using the attention mechanism adopted by the XLNet and XLCNN Transformer models.

1.4 Contribution

The main contribution of this thesis is the creation of XLCNN and how the scientific community but also cyber security researchers will be able to use its capabilities as antivirus software, as it can successfully identify malware with high rate. There is no need to apply a different algorithm in order to extract features from the data set, as the XLCNN was designed to accept the input data as it is and to select by itself the attributes that serve its purpose. It can also be used for general purpose problems, such as natural language processing, next sentence prediction, multi labeled classification, etc. The XLNet model was also used for the first time to detect malware in executable files. Although the results of XLNet were quite high, those of XLCNN were even higher, despite the fact that the data used for the training were smaller than those used by other researchers that use algorithms such as RNNs and LSTMs. This proves the effectiveness of XLNet and XLCNN models against other deep learning algorithms.

1.5 Related Work

There have been numerous attempts to detect malware based on Deep Learning, however, only one of them has used the Transformer Architecture. Namely, MALBERT [66] has used BERT [10] for Malware Detection in Android Systems. They achieved an accuracy of 97.61%. So we will elaborate on Malware Detection approaches based on Deep Learning and present only representative examples.

Hardy et al. [67] in DL4MD, based on the Windows Application Programming Interface (API) calls extracted from the Portable Executable (PE) files study how the Stacked Autoencoders (SAE) model can be designed for intelligent malware detection. The SAEs model employs greedy layerwise training operation for unsupervised learning, followed by supervised parameter fine tuning. They achieve an accuracy of 95.64% in the Testing phase.

In Rhode et al. [68] recurrent neural networks (RNNs) are used to predict whether or not an executable is malicious. They achieve 94% accuracy using 5 seconds of execution for each executable file. Pascanu et al. [69] also use RNNs for Malware Classification. Echo State Networks (ESNs) and Recurrent Neural Networks are used for the projection stage that extracts the figures. Echo State Networks have been successfully used for predicting chaotic systems. They achieve a true positive rate of 98.3% and a false positive rate of 0.1%. Kolosnjaji et al. [128] use Deep Learning for Classification of System Call Sequences based on data extracted from VirusTotal. They achieve an average of 85.6% on precision.

Finally, in the appendix of the Book [129] called “A Survey on Malware Detection from Deep Learning”, it is mentioned that the combination of a Convolutional Network with LSTM had an accuracy of 89.4%, the Feedforward Neural Network 79.8% and the Convolutional Network alone 89.2%.

1.6 Structure of the study

Chapter 2 presents the most well-known types of malware and analyzes the way in which they operate. Chapter 3 presents the two main categories, static and dynamic analysis, which analyzes the behavior of malware in a system. Chapter 3 analyzes various machine learning algorithms such as CNN, RNN and Transformers. Chapter 4 shows the methodology of XLNet, its architecture and compares it with other Transformer models such as BERT. Chapter 5 presents the basic architecture of XLCNN, the way in which training data was collected, the extraction of metadata from the executable files and finally the comparison of the results with XLNet.

2 Background

2.1 Malware types

A malware is defined as “any code added, altered or intentionally causing damage to a software system or to impair the intended function of the system” [70]. Specifically malware is part of a software or program that deliberately serves the malicious efforts of an attacker and it comes in many forms for different purposes. Common terms used to classify various types of malware are Trojans, worms, and viruses [24]. The fact that malware can cause loss of information, money and lives is a major threat to technological progress.

The classification of warehousing is based on the performance characteristics of the program. The malicious software is also classified based on its payload, how it exploits or weakens the system and how it spreads. This enables the discussed levers to be divided into different types as discussed below. Traditionally, they were developed to show one's strengths, for fun aspects, or to highlight weaknesses within the system. Today, however, these motivations go to the greatest betrayal. We can now see the spectrum of motivation from individual to national interest, and these days an entirely new underground economy is based on malware [25]. Below is a brief overview of the most popular types of malware

2.1.1 Virus

A virus is a self-replicating part of a malware. It exists as an executable file and is copied and distributed to other host systems. It is inactive and must be transmitted via files or media files or network files. Depending on the complexity of the code, it may modify copies of its own [71] viruses that can be used to damage the host computer and network, steal information, create botnets, advertise and steal money, among other malicious activities.

2.1.2 Trojans

A Trojan consists of two parts: a server that runs on the attacker's host and a client that runs on the attacker's console [74]. Server code (usually very small in size, some not larger than KB) is not sent to the infected person through some malware

distribution method. In a typical setting, the attacker sends a file containing the server code to the victim (for example, an image or PDF is large enough that the size of the server is smaller than the total size of the file). When the user double-clicks on the attacked file, they start the "Server" program included in the infected file. The server usually runs in stealth mode and is not easily visible to the user and / or the file manager. At this point, the server code in the infected file can communicate with the attacker's client code in a number of ways.

An easy way is to use a reverse connection in which the server code contains the IP address from which the attacker wants to control the victim's computer. But there are also much more sophisticated reverse connection methods. Once launched, the server program contacts the console with client-side code allowing the attacker to now control the victim's program. It can install new programs on the victim's computer (for example, keyloggers), it can read every file on the victim's computer (for example, credit card and bank information, personally identifying information), and more. Actually, he can control the victim's computer using his keyboard from a remote location.

In some cases, the Trojan is very obvious and makes many attempts to stay "out of sight". They take open control of the victim's car. However, a more dangerous situation also occurs when the Trojan is out of sight and operates in stealth mode for a long time, when the victim is unaware that his data (or his company's data) is being downloaded by an unscrupulous attacker.

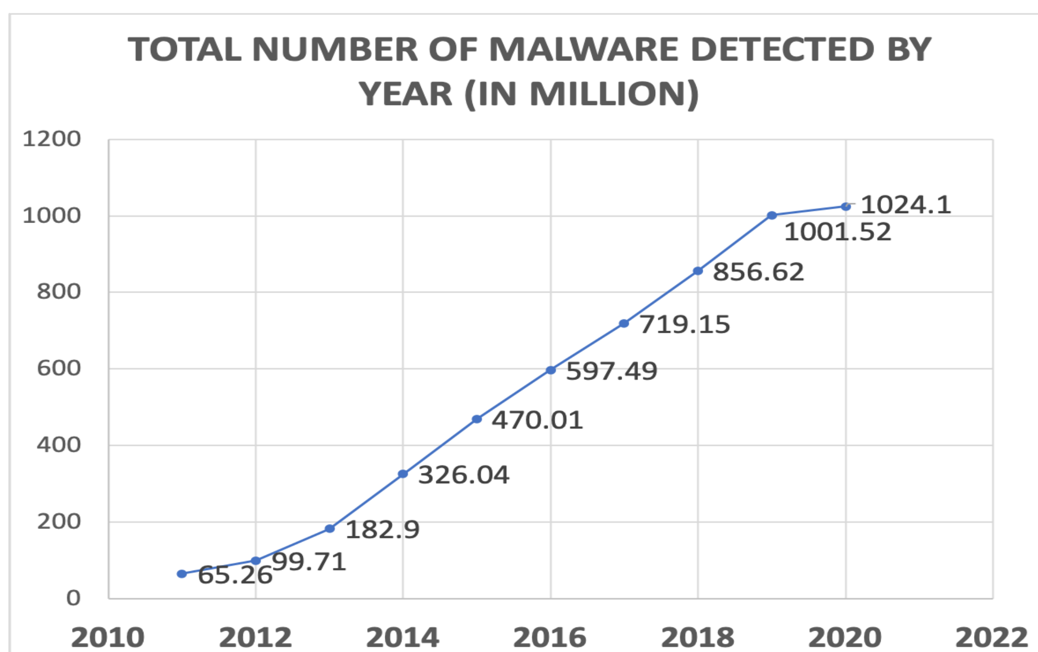


Figure 1. Total number of malware detected by year (in million) [75]

2.1.3 Worms

A worm is a piece of malware that can spread freely across a network by exploiting vulnerabilities in existing software to compromise a system. Worms can spread through networks in a number of ways. For example, worms can spread across a network by using email to infect other computers or by using other file transfer protocols to copy themselves to other systems. Worms can carry a payload. While some worms may do nothing but spread from one computer to another (simply using bandwidth and slowing down a network), others can do dangerous things like delete files on one computer or encrypt files so that the owner of the file must pay a ransom in order to decrypt his files. Weaver et all [76] divides the worms into 3 categories.

Targeting. This suggests the mechanism by which the worm is used to attack potential victims. Commonly used targeting mechanisms include scanning the network for vulnerable hosts, using specific target lists, using the "metaserver" (which is a regularly updated list of vulnerable servers) that the worm periodically queries to find new targets, and topological worms that discover the structure of a network in order to identify new targets, and “passive” worms that lie in wait for a target.

Distribution Mechanisms. Worms can spread in three ways. Self-generated worms spread freely (eg, topological worms and worms that spread through the network by scanning). Second Channel Worms are spread through an additional communication channel, such as a remote procedure call. Embedded worms spread by embedding themselves in a standard communication channel.

Activation Mechanism. Worms can be activated either by an explicit human action (e.g. via an infected email), an explicit human activity that is identified by the worm, triggering it, or injected into a host as part of a default process.

In general, topological worms and worms that propagate autonomously through scanning can be incredibly fast. Notorious computer worms include:

- **Stuxnet** [77] is probably the best-known example of a worm in recent years. Discovered by security vendor Kaspersky Labs in 2013, [77] reportedly launched by Israeli and US intelligence agencies Stuxnet was signed with a certificate stolen from two software makers in Taiwan. I did. Stuxnet was targeting the Natanz pyknosis

facility in Iran. The Stuxnet code has infected computers in several countries [78] but has not been reported to adversely affect SCADA systems other than Nutanz. Stuxnet launched the first social engineering attack to introduce a Stuxnet-infected Memory Stick. The worm spread rapidly. When infecting a host, Stuxnet first checked to see if it was a special type of Siemens equipment commonly used in nuclear facilities. In that case, the dropper program dropped the malicious code into the main () program loop of the Siemens controller. The malicious code contained a number of variants that targeted a particular type of controller.

- **Mydoom** [79] appear in emails with this message, prompting users (mostly Windows) to click on the attachment that affects their machine. Different versions of the Mydoom have different scales, one of which has a backdoor installed on the hunting machine so that the machine can be controlled remotely. Mydoom is thought to have used a large amount of Internet bandwidth when it targeted the Internet in 2004.
- **Morris worm** [80] were a mix of sophisticated and rustic. The overall design was simple. It checks your computer's system configuration to find, access, and reduce the number of breaks on any machine. The worm enabled the heuristic knowledge of Internet topology and trust to help spread and target two different machine architectures. While it is particularly effective to detect potential attack targets, it also requires time-consuming work of essaying passwords for individual user accounts, which is the "whole attack" aspect. Nevertheless, it was a sacrifice of its own success because it could not control exponential growth. The Morris worm was rampant because it had no global data or control points.

2.1.4 Adware

Adware, which is short for Advertising Supported Software, automatically delivers the ads that are seen in website pop-up ads and displayed by the software. Most are designed by advertisers to serve as a revenue generating tool. Some adware may be bundled with spyware, making it very dangerous as it can track user activity and steal user information [71].

2.1.5 Rootkit

This is a program that uses some tools to avoid detection on the system. These tools are highly invasive and difficult to remove because they are very sophisticated and complex programs created to hide within the legitimate process of infected computers. They are designed to give you full control of your system and get the highest possible permissions on your computer, among other possible malicious activities [81]. Due to the bypass technology used in rootkits, most security vendors' solutions are ineffective in detecting and removing them, so their detection and removal relies heavily on manual labor. These may include monitoring your computer system for anomalous activity, analyzing memory dumps, and scanning system file signatures.

2.1.6 Bots

Bots are programs designed to perform specific operations. Bots are derived from robots that were first developed to handle IRC-Internet relay chat channels, which appeared in 1989 as a text-based communication protocol [81]. Some bots are used for legitimate purposes such as video programming and other online competitions. Malicious bots are designed to create botnets. Botnets are defined as a network of host computers (zombies/bots) controlled by an attacker or botmaster. Bots infect other computers and infect other connected computers, thus creating a network of computers called botnets. Bots are commonly used as Spambots for DDOS attacks, scraping server data, distributing malware on WebSpiders, and downloading files from sites. Captcha tests are used to protect websites from being hacked by human users [81].

2.1.7 Ransomware

Ransomware is a type of malicious software that prevents users from accessing or blocking the system or files by locking the screen or encrypting files until the ransom is paid [82]. In most cases, ransomware leaves users with only a few options, which only allows the victim to communicate with the attacker and pay the ransom.

The most common types of ransomware use several types of encryption, including symmetric and public-key based encryption schemes. Ransomware that relies on public-key encryption is particularly difficult to mitigate, because encryption keys are

stored in remote command and control (C&C) servers. There is usually a time limit for paying the ransom, a special website for users to purchase cryptocurrencies (such as bitcoin) and step-by-step instructions on how to pay the ransom. The life cycle of modern-day ransomware typically consists of the following stages: distribution, infection, C&C communication, file discovery, file encryption, and ransom demand.

One of the most recent and popular ransomware was WannaCry [83]. The WannaCry ransomware (also known as Wana Decrypt0r, WCry, WannaCry, WannaCrypt, and WanaCrypt0r) was observed in a massive attack in several countries on May 12, 2017 [83]. According to various reports from security vendors, a total of 300,000 systems in more than 150 countries have been severely damaged. The attack affected a wide range of sectors, including healthcare, government, telecommunications, and oil/gas production. The image below shows the message displayed on the victim's computer screen by WannaCry.

Figure 2. Screenshot of the ransom note left on an infected system by WannaCry [83]

The difficulty of defending against WannaCry is due to its ability to spread to other systems using worm components. This feature requires a defense mechanism that makes the attack more effective and can react quickly and in real time. In addition, WannaCry has a cryptographic component based on public key cryptography.

During the infection phase, WanaCry uses Eternal-Blue and Double Pulsar exploits that were allegedly leaked in April 2017 by a group called Shadow Brokers. Eternal Blue exploited the threat of server message block (SMB) that Microsoft patched on March 14, 2017 and described in the security bulletin MS17-010 [84]. This threat allows opponents to execute remote codes by sending specially designed messages to the SMBV One server, connecting to TCP ports 139 and 445 Windows systems. In particular, this threat affects all non-structured versions of Windows, from Windows XP to Windows 8.1, except Windows 10.

DoublePulsar is a persistent backdoor that can be used to access previously compromised systems and run code so that the attacker can install additional malware on the system [83]. During the distribution process, the Wannakri worm component actively uses Eternal Blue for initial infection by SMB vulnerability by searching for suitable TCP ports and, if successful, attempts to implant doublepulsar backdoor on infected systems.

2.2 Malware analysis

Legitimate users are protected against malicious code by using antivirus software that identifies, analyzes it and alerts the user accordingly. Typically, an antivirus tool is equipped with a signature database that is used to identify potential known or common threats in the matching process. The malware analyst retrieves the suspicious piece of code and analyzes it to determine if it is harmful. When a threat is confirmed in the code, the analyst searches for a specific model of threat and develops a signature for that code (malware) and the signature is added to the database to deal with specific malware. Although this manual process may seem trivial and completing the task is time consuming and subject to errors as there are many variations of the same code. Statistics show that antimalware vendors encounter thousands of malicious codes every day.

Malware analysis has become an important and essential skill for security professionals and forensic investigators. Malware analysis not only enables the analyst to

understand the purpose of the malicious code, but also provides insight into the evolving trend of malware, providing analysts with a tool to improve their detection methods.

Stuxnet [26] is one of the latest images of such motivations. Malware advertises itself through an open or vulnerable network service using vulnerabilities in the targetted system, removable devices [26, 27] or through social engineering using a series of infectious agents. To combat malware, systems now have antivirus programs. Most antivirus programs include a scanner and a signature database. The scanner matches the file on the user's system and matches the available signatures. The alert is created and the user is notified when a match is found.

Typically, two methods are used to analyze malware. Static and dynamic analysis. The difference between the two methods is that dynamic analysis detects malicious behavior during the execution of the sample code, whereas the static approach does not execute the code.

2.2.1 Static Analysis

Analyzing a program without examining it to see its behavior is commonly known as static analysis. This can be done in a variety of ways, depending on the availability of the code and its presentation. Static analysis can help diagnose memory errors and improve program execution if its source code is available [28, 29]. It can also be used to perform a viable binary check with various tools [30]. Static analysis may be requested before or after dynamic analysis, or may be performed as a separate procedure. Sometimes it is done to see if the analysts have lost anything suspicious after the dynamic analysis. And the initial dynamic analysis is done to understand and understand this behavior before applying the code of conduct in the real environment.

Static analysis refers to scanning portable executable files (PE files) without executing them. Malware usually uses binary packers such as UPX and ASP pack shell to avoid scanning [32]. It must be unzipped before the PE file can be analyzed. The disassembly tool, such as IDA Pro and OlleyDbg, can be used to decompress a Windows executable file, display assembly instructions, provide information about malware, and extract the template to identify attackers.

Detective patterns can be tracked in static analysis such as Windows API calls, string signatures, control flow graphs (CFGs), C code (operation code) frequencies, and

byte sequences n-grams [33]. In the following, we explain the main features in static analysis. Almost all programs use the Windows API (short for Application Programming Interface) to communicate with the entire operating system. For example, “OpenFileW” in “Kernel32.dll” is a Windows API that creates a new file or opens an existing one.

Therefore, API calls reveal the behavior of the programs and can be considered as a necessary indication in the warehousing check. For example, the Windows API calls “WrightProcessMemory”, “LoadLibrary” and “CreateRemoteThread” are suspicious behaviors that are used by malware to process DLL injections, while rarely come together in a legal set. DLL injection is discussed in the Memory Analysis section. Wires are a good indicator of malicious existence. Strings reflect the intentions and goals of the attacker because they often contain serious semantic information [32]. For example, the following string "This program cannot be run in DOS mode" indicates a malicious file when it appears outside the typical PE header, a common feature of droppers and installers.

Control Flow Graph (CFG): A CFG is a directed graph that shows the control flow of a program. Blocks of code are represented by nodes, and control flow paths are represented by edges. If malware is detected, CFG can be used to track the behavior of PE files and extract the program structure [34].

Opcodes are the first part of a machine code notation (also called a machine language) that indicates which operations should be performed by the CPU. Whole machine language instructions with opcode and optional all, one or more operands (for example, “sub ebx 1”, “add eax ecx” and “mov eax 9”). Opcode can be used as a feature to detect malware by checking the opcode frequency or by calculating the similarity between op code sequences.

There are all the sequential effects of the sequence of n-gram [35]. For example, the word "MALWARE" is a sequence of 7 letters long, which can be divided into 3-grams: “MAL”, “ALW”, “LWA”, “WAR” and “ARE”. N-gram is implemented with various identification features such as API calls and opcodes. In addition to previous features, there are other features used in static analysis such as file size and function length. Networking features such as TCP / UDP ports, destination IP, and HTTP requests are static analysis features. Key research on malware signature theft techniques is Kirat and Vigna [36]. They were able to extract techniques from 2810 malware models and classify them into 78 types of stolen signature techniques.

Hashemi and Hamzeh presented a new approach that extracts a unique opcode from the executable file and converts it into a digital image. The visual features are then extracted from the image using the local binary pattern (LBP), which is one of the most popular texture extraction methods in image processing. Finally, machine learning methods are used to detect malware. The proposed detection technique obtained an accuracy rate of 91.9% [37]. Shaid and Maarof also suggested showing malware as images. Their technique captures malware API calls and converts them into visual signals or images [38].

On the other hand, Salehi et al. [39] and Han et al. [40] built their techniques based on the extracted API calls. Salehi et al. extract API calls from each binary file and uses API frequencies to learn the classifier. Then, three feature sets “API Call List”, “API Arguments” and “API and Argument List” were generated, and each set was tested separately. The results showed that the API argument list is better than the other two sets with an accuracy of 98.4% and a false positive rate of about 3%. Similarly, Han et al. Gets the API from the Import Address Table (IAT) using static analysis. They compared the extracted API sequence to another sequence and calculated the similarity between them to classify the malware family. Han et al. found that malware in the same family is about 40% identical, and the calculated false positive rate is 16%. Similarly, Cheng et al. [41] analyzed native API sequences using WinDbg tools and implemented Support Vector Machine to detect shellcode malware. They used a very small training set and were able to achieve an accuracy rate of 94.37%. However, the false negative rate was 44.44%. Table 1 shows the results of the surveyed papers who have made a comprehensive static analysis of their malware inspection methodology. The following sub-sections describe the related shortcomings of static analysis and the signature based analysis.

Table 1. Static analysis results for malware detection

Author Year	**Dynamic feature**	**Classifier**	**Datadet Malware/Benign**	**Acc**	FP
Liang 2016 [119]	API calls	DT, ANN, SVM	M=12,199	91.3%	-

Mohaisen 2013 [120]	File system, registry, network	SVM, DT, KNN	M=1,980	95%	5%
Mohaisen 2015 [121]	File system, registry, network	SVM, DT, KNN	M=115,000	99%	-
Galal 2017 [122]	APIs sequence	DT, RF, SVM	M=2,000/ B=2,000	97.2%	-
Ki 2015 [123]	APIs sequence	-	M=23,080	99.8%	0%
Fan 2015 [124]	User API, native API	J48, NB, SVM	M=773/B=253	95.9%	5%

2.2.1.1 Shortcomings of Static Analysis

Since the source code for most programs is not readily available, the static analysis approach is difficult to counter malware and reduces its application. Analyzing binaries with a static approach comes with complexity and challenges. Disassembly is an essential part of static binary analysis and can easily be obscured by simple obfuscation measures. Some malware with strong evasion and obfuscation techniques, such as the presence of opaque constants, obscures the disassembly of binary executables and cannot allow the analysis of the resulting code [31]. Such obfuscation techniques modify the program flow, make variables inaccessible, and disable tracking of values stored in registers. These limitations of the static approach motivate the development of analytical techniques that can overcome the breakouts and code transformations mentioned above and analyze malicious programs more accurately and reliably.

2.2.1.2 **Signature based Analysis**

The majority of available antivirus software uses a signature approach. This approach extracts a unique signature from the captured malware file and uses that signature to detect similar malware. The executable data bytes is first extracted and

then used as input for a hash function. The most common function used for such cases is called Sha-256 [55]. Its effect is used as an identification for this malware. Therefore, this method has a low false positive (FP) rate. However, it is not difficult for attackers to change the signature of the malware to avoid being detected by antivirus software. The signature database is much more efficient and faster in detecting known malware, but is unable to capture newly released malware [42]. The signature approach relies on the implementation of static analysis to extract stable byte sequences from malicious executable file. Figure 3 shows a general signature-based procedure for detecting malware.

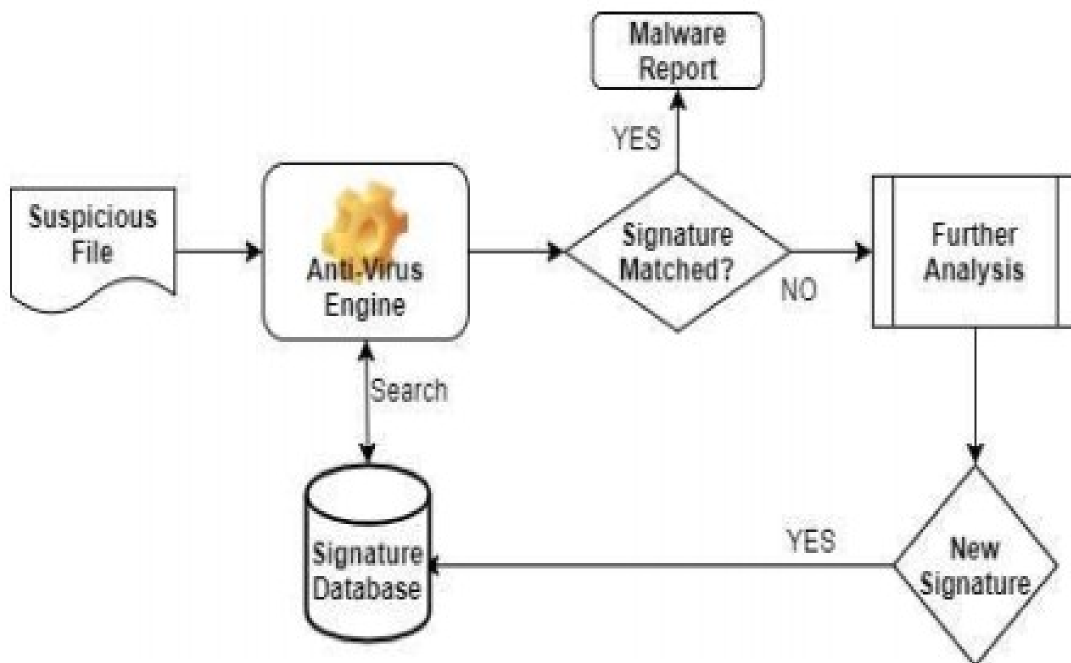


Figure 3: Signature-based procedure [42]

Shortcomings of signature-dased analysis have created another problem for signature-based approaches using transcription techniques. These techniques include dead code entry, register re-mapping, instruction replacement, and code manipulation [43].

- **Dead Code Insertion:** This simple code obfuscation technique changes the appearance by adding some no-op (NOP) statements or inserting ineffective PUSH / POP statements into the program, but keep its same behavior.
- **Register reallocation:** This technique works by switching registers or assigning register values to unused registers. For example, EAX is reassigned to the EBX register.
- **Subroutine reorganization:** Subroutines are a group of program operations that perform a particular task. This technique randomly changes the order of subroutines in a particular program.
- **Command replacement:** This technique replaces the original command that performs the same function with an equivalent command, such as replacing a MOV command with a PUSH command.
- **Code integration:** Malware code that has been incorporated into another legal program. To use this technique, the malware code decompiles the target program and inserts itself into the source code [44]. Code integration is considered one of the most advanced obfuscation techniques that allow malware to evade detection.

2.2.2 Dynamic Analysis

Dynamic Analysis [45] refers to the process of analyzing a code or script by running a code or script and observing its actions. These actions can be observed at different levels, from the lowest possible level (binary code itself) to the entire system (such as changes to the registry or file system). The purpose of dynamic analysis is to expose the malicious activity of performing an executable file while it is running, without compromising the security of the dynamic platform. From a defensive standpoint, malware must be loaded into RAM and run by a hosting CPU, so there is a risk of malware being infected during dynamic analysis.

Furthermore, dynamic analysis does not translate binary code into code at the assembly level. Although the disassembly process may seem straightforward, there are a number of techniques that attackers use to produce different assembly code from actually executing the disassembler program. By avoiding the disassembly process and not relying on the binary code of the parsed file, dynamic analysis is resistant to such

malware bypass techniques [46]. Although static analysis cannot detect changes made to code during execution, dynamic analysis is resistant to it.

The last comprehensive survey in the field of dynamic malware analysis was conducted in 2012 [46]. Since then, new types of malware (ransomware and cryptominers) and new analytical techniques (such as volatile memory forensics and side-channel analysis) have emerged. The survey was conducted to fill in the gaps and provide researchers with important information about the progress of the area.

Malware analysis is a structured process. The scan should begin by removing any wrapping that is applied to the binary code to hide from the scan tool (a process called unpacking [47, 48]). After receiving the unzipped file, a static analysis method can be applied to better understand the file. If the file matches the signature of the known malware, the scanning process can be skipped altogether. Therefore, static analysis is a fundamental step that reduces the need for further analysis. Depending on the analysis plan chosen for the task, other measures must be taken. Figure 4 shows a summary of the steps required for each analysis design, grouped according to their functionality. A detailed explanation follows.

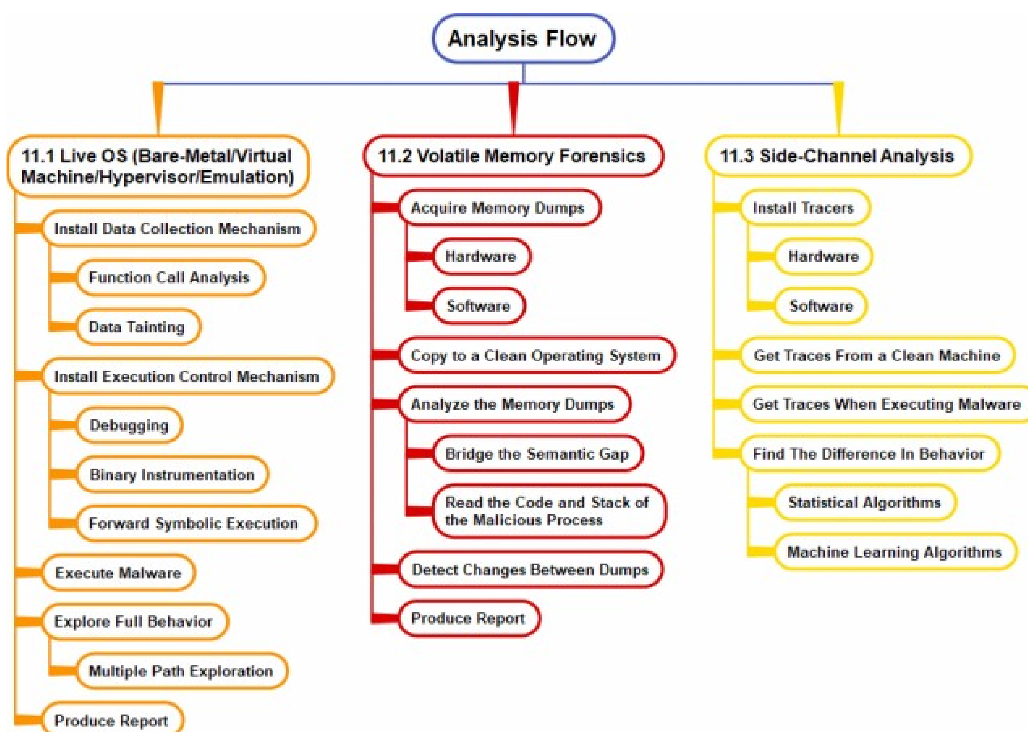


Figure 4: Dynamic malware analysis steps [46]

Most dynamic techniques focus on API calls that represent the behavior of malware. Liang et al. [49] introduced a behavior-based variant classification technique

that captures API calls from running malware and creates a multi-layered dependency chain based on call dependencies. This technique can measure the similarity between malware variants.

Unlike static analysis, dynamic analysis does not rely on binary code analysis and looks for meaningful patterns or signatures that indicate the maliciousness of the analyzed file. Such a static approach can be susceptible to many evasive techniques (e.g. packing, distortion, etc.). Compared to static scans, dynamic scans are more efficient because they do not require disassembling and scanning infected files. In addition, dynamic scanning can detect known and unknown malware. In addition, obscure and polymorphic malware cannot escape dynamic detection. However, dynamic analysis requires a lot of time and resources.

The following three sections will transform the three most common techniques used in dynamic malware analysis: function call monitoring, function hooking implementation and information flow tracking (IFT). Understanding these three methods provides a consistent, general picture of the malware executable. Functional call monitoring aims to investigate functional behavior, function hooking implementation gains full access to the actual arguments on the code stack and can perform the required analysis steps, while IFT also provides information on what data a malicious binary is interested in at runtime. Thus, the analyst has a clear idea of the behavior of the malware and how it affects the smooth operation of the system [23].

2.2.2.1 Function Call Monitoring

A function usually consists of code that performs a specific task, such as calculating a mathematical concept or creating a file. While the use of functions can lead to simple code reuse and easy maintenance, the property that makes the functions interesting for property analysis is that they are usually used for abstractions ranging from implementation details to semantic rich representations. For example, it doesn't matter if the result of a particular algorithm that applies the sort function matches the sorted input. When it comes to code analysis, such abstractions help you to get an idea of the behavior of the program. One way to keep track of what the program is saying is to block those calls. The process of intercepting function calls is called interception [50]. The process of the analyzed program takes place in such a way that in addition to the

intended function, the so-called interceptor function is called. This hook function is responsible for implementing essential parsing functions, such as writing a call to a log file or analyzing input parameters.

Application programming interface (API). Features that make up a consistent set of features, such as manipulating files and communicating over a network, are often grouped into so-called application programming interfaces (APIs). The operating system typically provides several APIs that an application can use to perform common tasks. These APIs are available at different layers of abstraction. Network access can be provided, for example, by an API that focuses on the content sent in TCP packets, or by a low-level API that allows an application to create a packet and write it directly to a raw socket. In the Windows operating system, the term Windows API refers to a set of APIs that provide access to various functional categories such as networking, security, system services, and management.

System calls. Software software running on computer systems, running shelf operating system products, is generally divided into two main parts. When normal applications, such as word processors or image manipulation programs, run in so-called user mode, the operating system runs in kernel mode. Only code that runs in kernel mode has direct access to system mode. This partition prevents user-mode processes from interacting directly with the system and its environment. For example, it is not possible to open or create a file directly for user space processing. Instead, the call operating system provides a particularly well-defined API: the system call interface. Through system calls, the user-mode application can request the operating system to perform limited actions on its behalf. Therefore, to create a file, the user-mode application needs to use a specific system call that describes the path, name, and access method of the file. After the system call is triggered, the system switches to kernel mode (that is, privileged operating system code is executed). By verifying that the calling genre application has sufficient access rights for the desired action, the operating system acts on behalf of the user-mode applications.

In the case of a file creation instance, the result of a system call is what is known as a file descriptor, where any other user mode application related to that file interacts (for example, writing to a file). This is done through manipulation. In addition to the full use of resources (within the limits of the operating system), a malware sample typically cannot operate within its limits. Therefore, malware (like any other application) that is

running in the user's space and needs to communicate with its environment should call the relevant system calls. Since system calls are the only way for a user to access or interact with their environment for mode operation, this interface is designed for dynamic malware analysis. Especially interesting. However, there are some well-known examples of malware that manipulate Daniel's methods [50]. Such events do not necessarily use the system call interface and can avoid this method of analysis.

Windows native API. The Windows Native API [51] resides between the system call interface and the Windows API. Although the Windows API remains constant for any version of the Windows operating system, the native API is not limited as such and may change with different service pack levels of the same Windows version. Native APIs are typically invoked from higher level APIs, such as the Windows API, to implement system calls and perform any necessary pre- or post-processing of arguments or results. Legitimate apps typically communicate with the operating system via the Windows API, but malicious code can bypass this layer and interact directly with native APIs to thwart monitoring solutions that only hook to Windows APIs. This of course comes with an additional burden on the malware author to design the malware to cover all the different versions of the native API. Since there is no official and comprehensive Native API documentation, it requires a lot of knowledge of Windows internals. Similarly, a malware author may decide to ignore the native API and invoke system calls directly from the malware. While this is possible, it requires an even less in-depth knowledge of the documented interface.

Results of Function Hooking. Hooking API tasks enables the analysis tool to monitor program behavior at the abstraction level of the related task. While meaningfully rich observations can be made by hooking up Windows API functions, a more detailed view of the same behavior can be obtained by observing the native API. The fact that the user must make system calls to interact with their environment through the space application suggests that this interface deserves special attention. However, this limitation only applies to malware that runs in user mode. A malware running in kernel mode can perform the desired tasks directly without going to the system call interface.

2.2.2.2 Function hooking implementation

Depending on the availability of the program's source code, different approaches can be used to hook functions. If the source code is available, call to hook functions can be inserted at appropriate places in the source code. Alternatively, compiler flags (eg -finstrument-functions in GCC [Free Software Foundation]) [52] can be used to implement hooks. Binary rewrite is used when the program to be analyzed is only available in binary form. To do this, two approaches can perform the necessary analysis.

- Rewrite the monitored function in such a way that the function calls the hook before executing its original code.
- Find and edit all call sites (ie call statements) when, when executed, hook up the monitoring function.

In both approaches, the hook function gains full access to the actual arguments on the stack and can perform the required analysis steps. Also, if the function is requested by a function pointer (for example, functions in shared libraries), this value can be changed to indicate the binding function. In the Windows operating system, the Detours Library function is available to facilitate call binding [50].

The idea behind Detours [53] is to use target function rewriting to implement interception of function calls. This is achieved by redirecting the control flow from the function to the analysis function, which in turn can call the original function. The control flow deviation is implemented by rewriting the original instructions of the original function with an unconditional jump in the analysis code. The overwritten instructions are saved and copied into the so-called trampoline function. This trampoline consists of secured instructions and an unconditional jump to the original function according to the overwritten instructions. As soon as the monitored application calls the linked function, new instructions redirect the control flow to the parsing code. This code can do any preprocessing (e.g. cleaning up arguments) and has full control over the control flow. The parsing code can then immediately return to the caller or call the original function by calling the trampoline. Since the original function is called by the parsing code, this code is monitored when the function returns and can then carry out any necessary post-processing.

Detours offers two alternative ways to apply the necessary modifications to the program. (1) it modifies the binary files before they are loaded and executed, or (2) it maintains an already-loaded binary in-memory image. The first technique requires adding additional partitions to a binary file while on disk. You need to modify the file's disk structure to accommodate additional code. The binary modification that is already executed is performed by DLL injection. First, all the payload (i.e. parsing functions) are compiled into a DLL. A new thread is created in the running binary that loads this DLL. When starting a DLL, the inventory handles the binary as described above (eg, create trampolines, overwrite the initial instructions of the target function).

Debugging techniques can also be used to invoke certain functions. Breakpoints can be added at the call site or in the monitoring function. When a breakpoint is triggered, the debugger who has full access to the memory contents and the processor status of the debugging process is given control. Thus, an instrumental debugger can be used to perform planned analysis.

If available, the link infrastructure provided by the operating system can be used to monitor system actions. The Windows operating system, for example, provides mechanisms for specifying messages and associated binding functions. Whenever an application receives a specific message (for example, a key was pressed on the keyboard or a button on the mouse was pressed), the hook function is executed. Changing dynamic shared libraries can serve as another means of monitoring function calls. For analysis purposes, native libraries can be renamed and replaced by stub libraries with link functions. These stubs can emulate the native behavior or call the native function in the library by renaming it. This method can fully capture the interaction of a program with a given API.

2.2.2.3 Information Flow Tracking

An orthogonal method of monitoring function calls while the program is running is an analysis of how the program processes data. The goal of information flow tracking as shown in Figure 3 is to shed light on the dissemination of “interesting” data across the system while a program that manipulates this data is running. Generally, the data to be monitored is labeled (contaminated) accordingly. The pollution label is promoted whenever data from the app is processed. For example, deployment statements usually

broadcast the source pollution label to the target. In addition to clear cases, policies that describe how pollution labels are promoted in more difficult situations should be implemented. In such a scenario, using the contaminated pointer as the source address evaluates to a contaminated value when indexing an array or conditional expression.

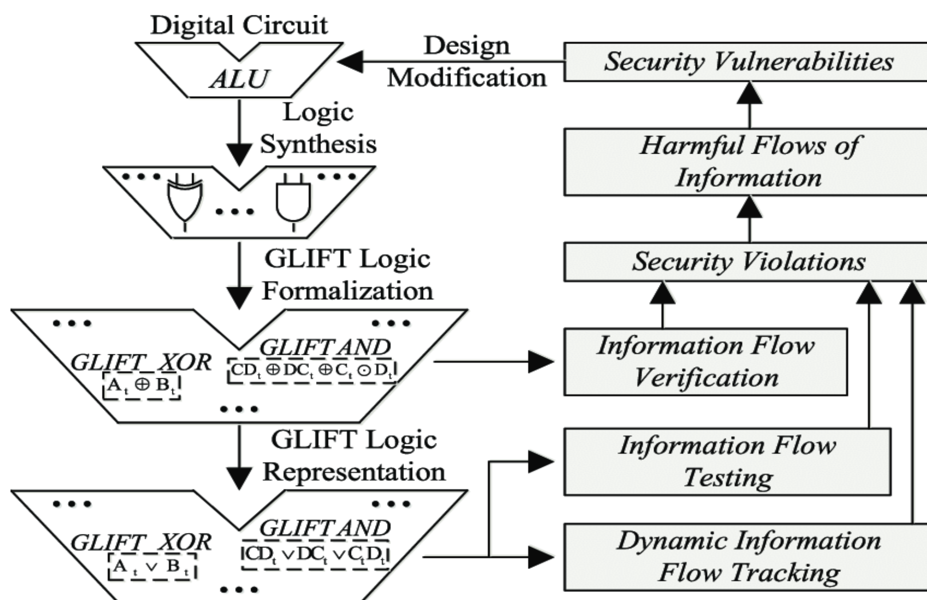


Figure 5. Information Flow Tracking [54]

2.3 Malware Detection

2.3.1 Windows PE

The PE file format was introduced as part of the original Win32 specification by Microsoft. However, PE files are derived from the previous Common Object File Format (COFF) found in VAX / VMS. Because there was a need for a common file format for all versions of Windows on all supported CPUs, it was decided to use the term "Portable Executable". To a large extent, this goal has been achieved in the same format used in Windows NT, Windows 95, and Windows CE [56]. Figure 1 shows the structure of the PE file compiled by the VC ++ compiler with C / C ++ language on the Win 32 platform.

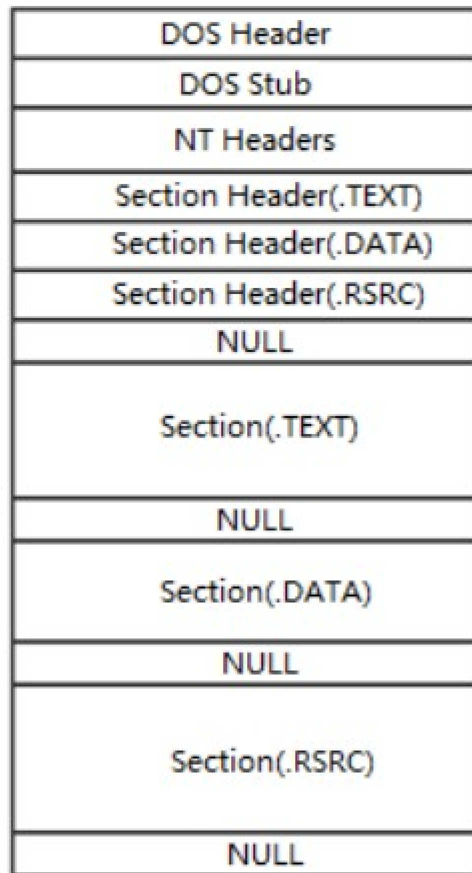


Figure 6: Figure 1. General C/C++ compiler PE file structure [57]

When an executable file is loaded into memory (for example, by calling LoadLibrary) then certain areas of a PE file are mapped to the address space. This results in the data structure being identical on disk and in memory. The bottom line is that if the user knows how to find an object in a PE file, it is almost certain that he can find the same information when the file is loaded into memory [56]. It is important to note that PE files are not simply mapped to memory as a single memory mapping file, but the Windows loader first examines the PE file and then decides which parts of the file will be assigned to the appropriate memory locations. Moving an item to the disk file may be different from moving it once it is loaded into memory. However, there is all the information that can be done to translate from disk offset to memory offset [56].

2.3.2 Obfuscation Techniques

Obfuscated malware is defined by structural and syntactic comparisons and variations of existing malware. These are divided into 3 groups. Packed, metamorphic and polymorphic malware depending on the detection technique used [71].

2.3.2.1 Packed Malware

Most malware authors repair or install multiple packages to create different versions of the same malware code. Perdisci, et al [72] claim that more than 80% of the new malware discovered are already packaged versions of existing malware. Packers shrink the file to a smaller size, and encryption is sometimes applied to compressed versions of the file to make the unpacking process easier. Some packers are customized by malware authors and can be used to determine if a file is compromised without the need for further analysis, but there are many commercial packers that are readily available online.

2.3.2.2 Oligomorphic Malware

Also called “semipolymorphic” malware [71], it uses several decryption procedures that are randomly selected during infection to avoid signature-based detection. The Whale virus was the first malware to use this technique, carrying dozens of different descriptors and choosing one at random.

2.3.2.3 Polymorphic Malware

Polymorphic malware, like oligomorphic malware, uses decryption procedures to change the appearance of runtime codes with each infection. They have a wide variety of decryption mechanisms as they tend to use mutation mechanisms. Mutation engines do all the logical calculations when reorganizing code to prevent signature match detection. The decryptor is executed first after the malware is copied to the machine, and it allows the malware to run. When the malware replicates, it encrypts the new malware with a different key and includes the new decryption procedure in the new code. However, it can only generate a few hundred decoders to be detected [73].

2.3.2.4 Metamorphic Malware

The body of the malicious code has been changed by a combination of different abominable techniques rather than appearance. Entering null bytes, reassigning registers, and transposing codes transforms the body of the code into a new generation, but works the same way [71]. That way, every change in malware generated looks different, so signature preparation and signature-based detection are very difficult. Unlike most polymorphic malware, which decrypts into a permanent body of code in memory, there may be different codes of metamorphic malware, which means that detection in memory is based on algorithmic scanning. Metamorphic malware can also inject and bind its code to the host program, making it difficult to detect malware.

3 Machine Learning

3.1 CNN

According to the Todd K. Barrett and David G. Sandier research work, there are many different neural network architectures [62]. The differentiation is located mainly by the way the processing nodes are interconnected, by the different calculation of arithmetic operations performed by each node, but also by the way it is chosen to train the neural network [62]. CNN exploits local correlations using local connectivity between nearby layer neurons [63]. As shown in Figure 7, the neurons in the m layer are connected to three adjacent neurons in the (m - 1) layer.

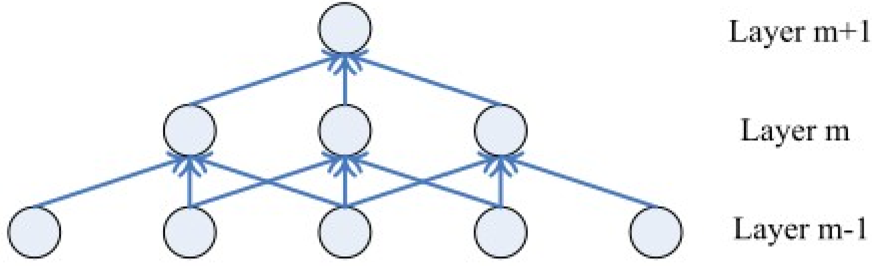


Figure 7: Local connections in the architecture of the CNN [63].

The value of a υ_x^{ij} neuron at position x of the *j*th feature map in the layer number i is denoted as follows [63]:

$$u_{ij}^{x}=g\left(b_{ij}+\sum_{m}\sum_{p=0}^{P_i-1} w_{ijm}^{p}\upsilon_{(i-1)m}^{x+p}\right) \tag{1}$$

where m indexes the feature map in the previous layer (*(i − 1)*th layer) connected to the current feature map, w_{ijm}^{p} is the weight of position p connected to the mth feature map, P_i is the width of the kernel toward the spectral dimension, and b_{ij} is the bias of *j*th feature map in the *i*th layer [63].

The output of the convolutional layer is usually the input to a pooling layer. The purpose of the concentration layers is to gradually reduce the spatial area (height, width) of the matrix without affecting its depth. In this way the number of trainable parameters in the network is reduced and consequently the computational cost is reduced by always checking the overfitting [64].

3.2 RNN

Due to increasing computing resources, recurring neural networks (RNNs), which have been around for decades but their full potential has only recently begun to be widely recognized such as convolutional neural networks (CNNs), have recently created significant growth in deep learning. In recent years, RNNs have played an important role in the fields of computer vision, natural language processing (NLP), semantic comprehension, speech recognition, language modeling, translation, image description, and human action recognition.

In prior studies, a number of approaches based on traditional machine learning, including SVM [58], K-Nearest Neighbour (KNN) [59] , Random Forest (RF) [60] and others, have been proposed and have achieved success for a malware detection system. In recent years, deep learning, a branch of machine learning, has become increasingly popular and has been applied to malware detection. Studies have shown that deep learning goes far beyond traditional methods. Most authors use a deep learning approach based on a deep neural network to detect flow-based abnormalities, and experimental results show that deep learning can be applied to malware detection.

Recurrent neural networks include input units, output units, and hidden units, and the hidden unit completes the most important task. The RNN model has essentially a one-way flow of information from the input modules to the hidden modules, and the composition of the one-way information flow from the previous temporal module to the current timing module is shown in Figure 8. The hidden modules can be considered as the storage of network, remembering information from end to end. When we unfold the RNN, we can see that it incorporates deep learning. An RNNs approach can be used for supervised classification learning.

Recurrent neural networks have introduced a directional loop that can memorize previous information and apply it to the current output, which is the essential difference from traditional Feed-forward neural networks (FNNs). The previous output is also related to the current output of a sequence and the nodes between the hidden levels are no longer connected. Instead, they have connections [61]. Not only the output of the input layer but also the output of the last hidden layer acts at the input of the hidden layer.

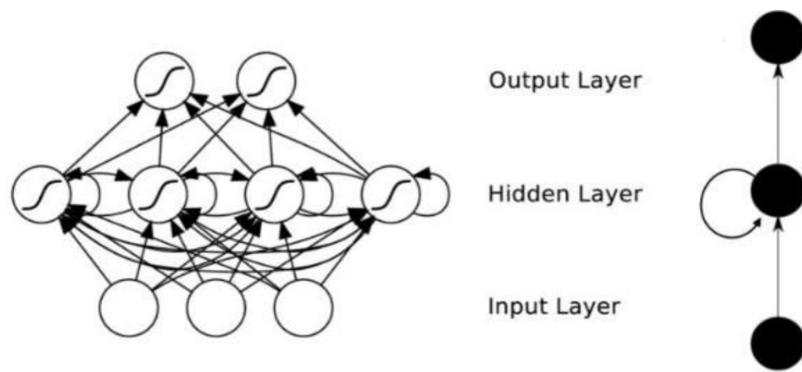


Figure 8: Recurrent Neural Network (RNNs) Architecture

3.3 Transformers

Recurrent models usually contribute primarily to the mathematical calculation along the symbol positions of the input and output inputs. They create a hidden position sequence, as a function of the previous hidden state - 1 and the input for the position, using alignment of the positions in steps at the calculation time. This inherently later nature results in the divergence of parallelism in the mathematical operations performed in the training examples. This is a problem when there are long lengths, as memory constraints limit the total computation time. By applying the Transformer model architecture, significant improvements in computational efficiency are achieved through factorization tricks [7] and under computational conditions [8], while at the same time improving the performance and efficiency of the model. However, the basic restraint of sequential calculation continues to exist.

The attention mechanisms [5] are now the most powerful and effective tool for modeling sequences and conversion models in various tasks, as without taking into account their distance in the input or output sequences, they allow the modeling of dependencies. However, in many cases, these attention mechanisms are used in conjunction with an iterative network with the goal of improving neural network performance. The attention mechanism [5] directs a neural network to focus on those features that are important for achieving a goal, which can be a problem of sorting, translating or even predicting future values. There is a wide range of applications that include sequence-based models that focus on deep neural networks. In the following

sections, I will describe the Transformer, get its attention and discuss its advantages over other models.

3.3.1 Attention Mechanism

The foundations of the extended neural GPU [11], ByteNet [12] and ConvS2S [13], which use convergent neural networks as the basic building block, while calculating hidden representations for all input and output positions, are to reduce the sequential computation. In these models, the number of functions required to associate signals from two arbitrary input or output locations increases in distance between locations, linearly for ConvS2S and logarithmically for ByteNet, making it more difficult to learn dependencies between remote locations [5]. Although at a cost of reduced effective resolution due to weighted average focus positions, the Transformer architecture is reduced to a fixed number of functions.

Self-attention, sometimes called intra-attention, is a mechanism of attention associated with different positions of an individual sequence in order to compute a representation of the sequence. Self-attention has been used successfully in a variety of tasks, such as abstract summarizing, text engaging, reading comprehension, and learning independent sentence representation tasks. End-to-end memory networks rely on a repetitive attention mechanism instead of a repetitive sequence, and have been shown to perform well in simple languages, language modeling tasks, and answering questions.

The Transformer is the first switching model that without the use of RNN or sequence-aligned sequence relies solely on self-attention to calculate its input and output representations. A function of attention is to map a query and arrange a set of key value pairs in the output, where query, keys, values and output are all vectors. The output is calculated by the weighted portion of the values, where the weight assigned to each key is correlated with the query match function with that key.

3.3.2 The Transformer model architecture

If not all, then most competing neural sequence switching models have a encoder-decoder architecture as shown in the figure 9 below . Here, the encoder maps a sequence of continuous representations $z = (z_1,. .., z_n)$ to an input representation sequence of symbols $(x_1, ..., x_n)$. Since, the decoder then generates an output $(y_1, ..., y_m)$ of symbols

one element at a time. Consumes the previously created symbols as an additional input when creating the next one, while at each step the model applies automatic regression.

The Transformer model architecture uses stacked self-attention and perfectly interconnected layers and follows this overall architecture for both the decoder and the encoder, shown in the left and right halves of Figure 9, respectively. Transformers consists of two basic features, encoder and decoder.

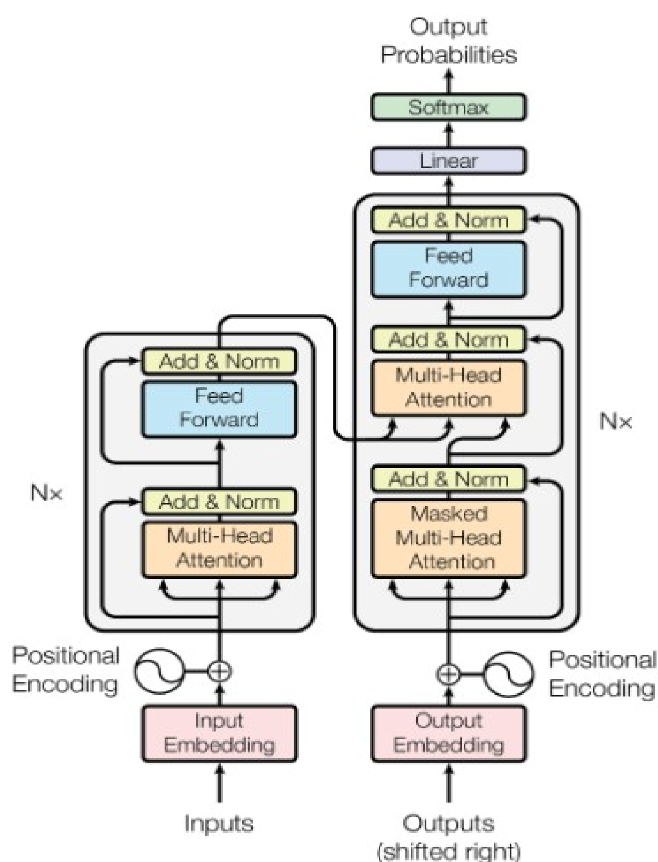
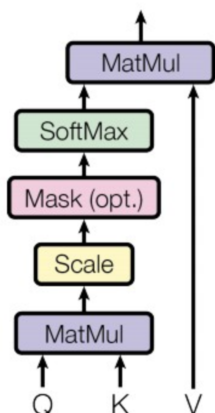


Figure 9. The Transformer - model architecture.

The encoder is made up of a stack of n = 6 equal layers. Each level has two sublayers. The first is a multi-head self-focus mechanism, and the second is a simple feedforward network that is fully connected according to position. A residual attachment is used around each of the two sublayers and then and then normalization is applied at all layers [14]. The output of each sublayer is called layer normalization (x + sublayer (x)),

whereas sublayer (x) is a function executed by the sublayer itself. To facilitate these residual connections, all sublayers of the model, as well as the embedding layers, produce an output of DimensionModel = 512.

The decoder also has a set of equal levels n = 6. In addition to the two sublayers in each encoder layer, the decoder inserts a third sublayer that pays multiple attention to the output of the encoder stack. Like the encoder, residual connections are used around each sublayer, followed by level normalization. The self-focus sublayer in the decoder stack is also modified so that the position does not go into the subsequent position. This masking, combined with the fact that the output structure is compensated by a condition, ensures that the prognosis for the positivist is based only on the known output at a position less than i.



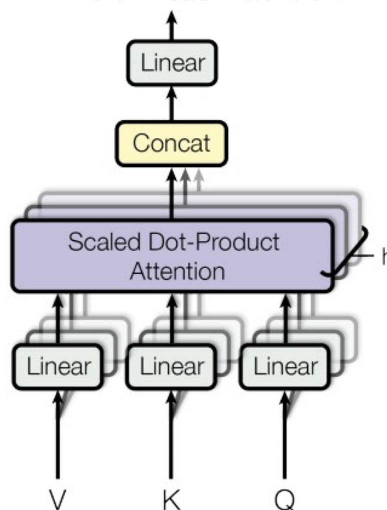


Figure 10. (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of severalattention layers running in parallel.

3.3.3 Scaled Dot-Product Attention

The special focus used by the Transformers is called "scaled dot product focus" (Figure 10). Input includes dimension d_k queries and keys and dimension d_v values. It is calculated the query product with all the keys, distribute each via $\sqrt{d_k}$, and applied the

softmax function to weigh the values. In practice, a focus function is calculated on a Matrix Q of questions simultaneously. Keys and values are also packed together in matrix K and V. The attention matrix of Q, K, V is calculated as:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V \qquad (2)$$

The two most commonly used attention functions are additive attention and dot product (multiplicative) attention. The scalar product attention is identical to our algorithm, except for the scale factor of $\frac{1}{\sqrt{d_k}}$. Extra attention calculates the compatibility function using a feedback network with a single hidden layer. Although the two are similar in theoretical complexity, dot product attention is much faster and more space efficient in practice, as it can be implemented using highly optimized matrix multiplication code. disregards the unscaled scaled product for values greater than dk [15]. We suspect that for large values of d_k, the dot products become large in magnitude, pushing the softmax function towards regions where it has extremely small gradients. To counteract this effect, we scale the scalar products of $\frac{1}{\sqrt{d_k}}$.

3.3.4 Multi-Head Attention

Rather than performing a single attention function with keys, values, and d_{model} dimension queries, we have found it useful to linearly project queries, keys, and h-values with different linear projections learned on the d_k dimensions, d_k and d_v , respectively. Then, for each of these predicted versions of queries, keys, and values, we run an attention function in parallel, producing two-dimensional outputs. They are combined and re-projected, which gives the final values, as shown on the right of Figure 10.

Multi-head focusing allows the model to jointly participate in information from different representation sub-locations in different positions. With a single focus head, the average prevents this.

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O \qquad (3)$$

where $head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$,

where the projections are parameter matrices $W_i^Q \in \Re^{d_{model} \times d_k}, W_i^K \in \Re^{d_{model} \times d_k}, W_i^V \in \Re^{d_{model} \times d_v}$.

3.3.5 Embeddings and Softmax

Like other sequential transduction models, transformers use the embedding learned to convert input tokens and output tokens to dimensional modal vectors. The quotient is using the commonly learned linear transformation and the Softmax function to convert the output to the next token possibility. Transformers share the same weight matrix between both the embedded layers and the pre-softmax linear transformation, as in [17]. In embedded layers, the weight is multiplied by $\sqrt{d_{model}}$.

4 Transformer Models

4.1 XLNet

XLNet [6] is the latest and greatest model emerging from the evolving field of Natural Language Processing (NLP). XLNet [6] paper combines recent developments in NLP with innovative options for how to approach the language modeling problem. When trained in a very large data set of words, the XLNet model achieves state-of-the-art performance for standard NLP tasks that belongs to the GLUE [126] benchmark. XLNet is an automatic language regression model that extracts the common probability of a sequence of badges based on the iterative transformer architecture. Its educational goal calculates the probability of a word sign that depends on all variants of the word signs in a sentence, as opposed to those on the left or only those to the right of the sign.

Unsupervised representational learning has been particularly successful in the field of natural language processing [18, 19, 20]. Typically, these methods pre-generate neural networks in a large-scale unlabeled text company and then optimize models or representations in later work. In the context of this common high-level idea, different unsupervised pre-training objectives were explored in the literature. Among them, self-aggressive language modeling and automatic coding (AC) were the two most successful preparatory goals.

By comparison with classical deep learning methods aggregate expenditure (AE)-based pre-training does not estimate density but aims to reconstruct the original data from damaged input. A notable example is BERT [10], which was the advanced pre-workout approach. Given the sequence of input badges, a specific part of the badges is replaced by a special symbol [MASK] and the model is trained to recover the original badges from the damaged version. As density estimation is not part of the objective, BERT is permitted to use two-way frames for reconstruction. As a direct benefit, this closes the aforementioned two-way information gap in autoregressive (AR) language modeling [21], leading to improved performance. However, artificial symbols such as [MASK] used by BERT during pre-training are absent from the actual data at the time of refinement, resulting in pre-processing mismatch. In addition, since the provided insignia are covered at the input, BERT is not able to model the possibility of sharing using the product rule as in the AR language model. In other words, BERT assumes that the

provided tokens are independent of each other, since undeciphered tokens are multiplied as long-range, long-range dependency is more prevalent in natural language [22].

4.1.1 Characteristics

In this section it is analyzed XLNet, a generalized autoregressive model that leverages the best of both AR language modeling and AE while avoiding their limitations.

- First, instead of using a fixed forward or backward engagement sequence as in conventional AR models, XLNet maximizes the expected probability of logging a w.r.t. sequence according to the order of factorization of all possible permutations. Thanks to the camouflage function, the environment for each position can consist of chips from left and right. In anticipation, each position learns to use contextual information from all positions, that is, by recording a two-way context [22].

- Second, as a generalized AR language model, XLNet is not based on data corruption. Therefore, XLNet does not suffer from the pre-treatment-final coordination mismatch to which BERT is subject. Meanwhile, the self-aggression goal also provides a natural way of using the product rule to factorize the common probability of the intended brands, eliminating the independence case made at BERT. In addition to a new pre-training goal, XLNet is improving architectural plans for pre-training.

- Inspired by the latest developments in AR language modeling, XLNet incorporates the Transformer-XL partitioning mechanism and related Transformer-XL coding scheme into pre-training, which empirically improves performance specifically for tasks involving a larger text sequence [6].

- Naively applying a Transformer (-XL) architecture to variant-based language modeling does not work because the ordering factor is arbitrary and the target is ambiguous. As a solution, it s proposed to redefine the Transformer network (-XL) [127] to remove the ambiguity. Empirically, under comparable experiment setup, XLNet consistently outperforms BERT [14] in a wide range of tasks such as GLUE

language comprehension tasks, reading comprehension tasks such as SQuAD and RACE, text sorting tasks such as Yelp and IMDB-B work.

4.1.2 Architecture

Figure 11: (a): Content stream attention, which is the same as the standard self-attention. (b): Query stream attention, which does not have access information about the content x z t . (c): Overview of the permutation language modeling training with two-stream attention.

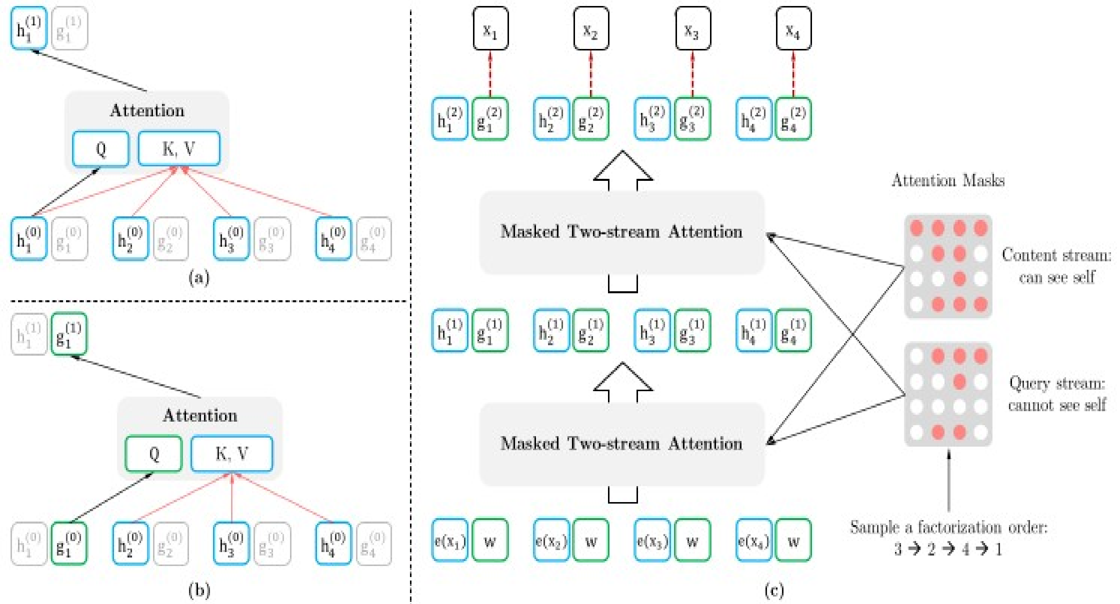


Figure 11. XLNet Architecture [6]

While the permutation language modeling objective has desired properties, naive implementation with standard Transformer parameterization may not work. To see the problem, assume we parameterize the next-token distribution p_θ (X_{Zt} = x | $X_{z<t}$) using the standard Softmax formulation

$$p_\theta(X_{Zt}=x|X_{Z<t})=\frac{\exp(e(x)^T g_\theta(X_{Z<t},Z_t))}{\sum_{x'}\exp(e(x')^T g_\theta(X_{Z<t},Z_t))}, \quad (4)$$

where g_θ ($x_{z<t}$) denotes the hidden representation of $x_{z<t}$ produced by the shared Transformer network after proper masking. Now notice that the representation h_θ ($x_{z<t}$) does not depend on which position it will predict, i.e., the value of z_t . Consequently, the same distribution is predicted regardless of the target position, which is not able to learn useful representations. To avoid this problem, suggested by the authors of XLNet to re-parameterize the next-token distribution to be target position aware:

$$p_\theta\left(X_{Zt}=x \middle| X_{Z<t}\right)=\frac{\exp\left(e(x)^T g_\theta\left(X_{Z<t}, Z_t\right)\right)}{\sum_{x'} \exp\left(e(x')^T g_\theta\left(X_{Z<t}, Z_t\right)\right)}, \quad (5)$$

where $g_\theta(x_{z<t}, z_t)$ denotes a new type of representations which additionally take the target position z_t as input.

4.1.3 Two-Stream Self-Attention

While the idea of target-aware representations removes the ambiguity in target prediction, how to formulate g_θ ($x_{z<t}$, z_t) remains a non-trivial problem. Among other possibilities, we propose to “stand” at the target position z_t and rely on the position z_t to gather information from the context $x_{z<t}$ through attention. For this parameterization to work, there are two requirements that are contradictory in a standard Transformer architecture:

(1) to predict the token x_{z_t}, g_θ ($x_{z<t}$, z_t) should only use the position z_t and not the content x_{z_t} , otherwise the objective becomes trivial,

(2) to predict the other tokens x_{zj} with j > t, g_θ ($x_{z<t}$, z_t) should also encode the content x_{zt} to provide full contextual information. To resolve such a contradiction, suggested from thw authors to use two sets of hidden representations instead of one:

- The content representation h_θ ($x_{z\le t}$), or abbreviated as h_{Zt} , which serves a similar role to the standard hidden states in Transformer. This representation encodes both the context and x_{Zt} itself.

- The query representation $g_\theta(x_{z<t}, z_t)$, or abbreviated as g_{z_t}, which only has access to the contextual information $x_{z<t}$ and the position z_t, but not the content x_{Z_t}, as discussed above.

Computationally, the first layer query stream is initialized with a trainable vector, i.e. $g_i^{(0)} = w$, while the content stream is set to the corresponding word embedding, i.e. $h_i^{(0)} = e(x_i)$. For each self-attention layer m = 1, . . . , M , the two streams of representations are schematically updated with a shared set of parameters as follows (illustrated in Figures 1 (a) and (b)):

$g_{z_t}^{(m)} \leftarrow$ Attention($Q = g_{z_t}^{(m-1)}$, $KV = h_{z<t}^{(m-1)}$; θ), (query stream: use z_t but cannot see x_{z_t})

$h_{z_t}^{(m)} \leftarrow$ Attention($Q = h_{z_t}^{(m-1)}$, $KV = h_{z<=t}^{(m-1)}$; θ), (query stream: use both z_t and x_{z_t})

where Q, K, V denote the query, key, and value in an attention operation [5]. The update rule of the content representations is exactly the same as the standard self-attention, so during finetuning, we can simply drop the query stream and use the content stream as a normal Transformer(-XL).

4.1.4 Partial Prediction

While modulation language modeling has many benefits, it is a much more difficult optimization problem due to variation and causes slow convergence in preliminary experiments. To reduce the difficulty of optimization, was chosen to predict only the last insignia in order of factorization. Formally, we split z into a non-target subsequence $z \leq c$ and a target subsequence $z > c$, where c is the cutting point. The objective is to maximize the log-likelihood of the target subsequence conditioned on the non-target subsequence, i.e.,

$$max_\theta E_{Z \sim ZT} \left[\log \left(p_\theta \left(X_{Z>c} \middle| X_{Z \leq c} \right) \right) \right] = E_{Z \sim Z_T} \left[\sum_{t=c+1}^{|z|} \log p_\theta \left(x_{Zt} \middle| x_{Z<t} \right) \right] \tag{6}$$

Note that $z > c$ is chosen as the target because it possesses the longest context in the sequence given the current factorization order z. A hyperparameter K is used such that about 1/K tokens are selected for predictions; i.e., $|z|/(|z| - c) \approx K$. For unselected tokens, their query representations need not be computed, which saves speed and memory.

4.1.5 Relative Segment Encodings

Architecturally XLNet differents from BERT which adds a complete segment integration to the word integration in each location, extending the idea of the relevant encodings from Transformer-XL to also codify the components. Given a pair of positions i and j in the sequence, if i and j are from the same segment, segment coding is used $s_{ij} = s_+$ or otherwise $s_{ij} = s_-$, where s_+ and s_- are learnable model parameters for each attention head. In other words, the purpose is to examine only whether the two positions are in the same department, as opposed to examining which specific departments they come from.

This is in line with the basic idea of coding. that is, by modeling only the relationships between positions. When i attends to j, the segment encoding s_{ij} is used to compute an attention weight $a_{ij} = (q_{i+b}) > s_{ij}$, Where q_i is the query vector as a standard attention operation and b is the learnable head-specific bias vector. Finally, the value of a_{ij} is added to the normal attention load. There are two advantages to using relative clause encoding. First, the inductive bias of relative encoding improves generalization. Second, it opens up the possibility of fine-tuning functions with three or more input segments. This is not possible with full segment encoding.

4.1.6 Comparative analysis

XLNet used BooksCorpus and English Wikipedia as part of preparatory data, which combines plain 13 GB text. In addition, Giga5 (16 GB text), ClueWeb 2012-B and Common Crawl for pre-training are included. XLNet’s authors used heuristically to filter aggressively short or low quality articles for ClueWeb 2012-B and Common Crawl, which leads to 19 GB and 110 GB text respectively. After tokenization with SentencePiece, it acquired 2.78B, 1.09B, 4.75B, 4.30B and 19.97B subword tracks for Wikipedia, BooksCorpus, Giga5, ClueWeb and Common Crawl respectively, which total 32.89B.

The XLNet-Large model has the same architecture hyperparameters as the BERT-Large, resulting in a similar model size. During the pre-training, a full sequence length of 512 was used. First, to provide a fair comparison with BERT, XLNet-Large-wikibooks were also trained only in BooksCorpus and Wikipedia, where all pre-preparatory hyper-parameters such as in the original BERT. Next, increase the training of XLNet-Large using all the data sets described above. Specifically, the model was trained in 512 TPU v3 chips for 500K steps with Adam weight splitting optimizer, linear learning rate splitting and batch size 8192, lasting about 5.5 days. It was observed that the model still lags behind the data at the end of the training.

Since the recurrence mechanism is introduced, authors used a bidirectional data input pipeline where each of the forward and backward directions takes half of the batch size. For training XLNet-Large, it was set the partial prediction constant K as 6. The finetuning procedure follows BERT [10] except otherwise specified 3 . It was employed an idea of span-based prediction, where we first sample a length L $\in [1, \cdots, 5]$, and then randomly select a consecutive span of L tokens as prediction targets within a context of (KL) tokens.

Table 2. Fair comparison with BERT. All models are trained using the same data and hyperparameters as in BERT. We use the best of 3 BERT variants for comparison; i.e., the original BERT, BERT with whole word masking, and BERT without next sentence prediction.

Model	SQuAD1.1	SQuAD2.0	RACE	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B
BERT-Large (Best of 3)	86.7/92.8	82.8/85.5	75.1	87.3	93.0	91.4	74.0	94.0	88.7	63.7	90.2
XLNet-Large-wikibooks	88.2/94.0	85.1/87.8	77.4	88.4	93.9	91.8	81.2	94.4	90.0	65.2	91.1

Table 3. Comparison with state-of-the-art results on the test set of RACE, a reading comprehension task, and on ClueWeb09-B, a document ranking task. ∗ indicates using ensembles. indicates our implementations. “Middle” and “High” in RACE are two subsets representing middle and high school difficulty levels. All BERT, RoBERTa, and XLNet results are obtained with a 24-layer architecture with similar model sizes (BERT-Large).

RACE	**Accuracy**	**Middle**	**High**	**Model**	**NDCG@20**	**ERR@20**
GPT	59.0	62.9	57.4	DRMM	24.3	13.8
BERT	72.0	76.6	70.1	KNRM	26.9	14.9
BERT+DCMN	74.1	79.5	71.8	Conv	28.7	18.1
RoBERTa	83.2	86.5	81.8	BERT†	30.53	18.67
XLNet	**85.4**	**88.6**	**84.0**	XLNet	**31.10**	**20.28**

Table 4. Comparison with state-of-the-art error rates on the test sets of several text classification datasets. All BERT and XLNet results are obtained with a 24-layer architecture with similar model sizes (BERT-Large).

Model	IMDB	Yelp-2	Yelp-5	DBpedia	AG	Amazon-2	Amazon-5
CNN	-	2.90	32.39	0.84	6.57	3.79	36.24
DPCNN	-	2.64	30.58	0.88	6.87	3.32	34.81
Mixed VAT	4.32	-	-	0.70	4.95	-	-
ULMFiT	4.6	2.16	29.98	0.80	5.01	-	-
BERT	4.51	1.89	29.32	0.64	-	2.63	34.17
XLNet	**3.20**	**1.37**	**27.05**	**0.60**	**4.45**	**2.11**	**31.67**

Table 5. Results on GLUE. * indicates using ensembles, and † denotes single-task results in a multi-task row. All dev results are the median of 10 runs. The upper section shows direct comparison on dev data and the lower section shows comparison with state-of-the-art results on the public leaderboard.

Model	MNLI	QNLI	QQP	RTE	SST-2	MRPC	CoLA	STS-B	WNLI
BERT	86.6/-	92.3	91.3	70.4	93.2	88.0	60.6	90.0	-
RoBERTa	90.2/90.2	94.7	92.2	**86.6**	96.4	**90.9**	68.0	92.4	-
XLNet	**90.8/90.8**	**94.9**	**92.3**	85.9	**97.0**	90.8	**69.0**	**92.5**	-
Multi-task ensembles on test (from leaderboard as of Oct 28, 2019)									
MT-DNN*	87.9/87.4	96.0	89.9	86.3	96.5	92.7	68.4	91.1	89.0
RoBERTa*	90.8/90.2	98.9	90.2	88.2	96.7	92.3	67.8	92.2	89.0
XLNet*	**90.9/90.9**†	**99.0**†	**90.4**†	**88.5**	**97.1**†	**92.9**	**70.2**	**93.0**	**92.5**

For explicit reasoning tasks like SQuAD and RACE that involve longer context, the performance gain of XLNet is usually larger. This superiority at dealing with longer context could come from the Transformer-XL backbone in XLNet. For classification tasks that already have abundant supervised examples such as MNLI (>390K), Yelp (>560K) and Amazon (>3M), XLNet still lead to substantial gains.

4.2 Other Transformer Models

Following the initial publication of XLNet, several other prefabricated models were released, such as the RoBERTa [16] and the ALBERT [17]. Since ALBERT implies an increase in the hidden model size from 1024 to 2048/4096 and therefore significantly increases the number of calculations in FLOPs terms, ALBERT excluded from the following results, as it is difficult to draw scientific conclusions. To get a fair comparison

with RoBERTa, the experiment in this section is based on complete data and reuses RoBERTa hyper-parameters.

5 The proposed malware detection approach

5.1 Backround Architecture

5.1.1 Sequence-to-Sequence

Typically, a neural network for sequence modeling of classification problems consists of the following layers:

Embedding layer. In this layer the model receives a sequence of words and tries to predict the probability of the next word appearing [86]. Specifically the input array, which is a loosely-encoded (e.g., hot-encoded) input token, is mapped to a denser feature layer. This is necessary because a high-dimensional feature vector is more capable of encoding information about a particular token (the term for the text corpus) than a simple hot-encoded vector. Instead of a pre-trained vector of words being used in this work, a state of the art tokenizer [96] was deployed specially made for the classification of malicious applications using their metadata.

Encoder Layer. After mapping the input multidimensional token, the sequence passes through the encoder layer to compress all the information from the input embedding layer (the whole sequence) into a specific vector of a fixed length. The encoder is made up of a stack of n = 24 equal layers. Each level has two sublayers. The first is a multi-head self-focus mechanism and the second is a simple feedforward network that is fully connected according to position. A residual attachment is used around each of the two sublayers followed by normalization [14]. That is, the output of each sublayer is called layer normalization (x + sublayer(x)), whereas sublayer(x) is a function executed by the sublayer itself. To facilitate these residual connections all sublayers of the model, as well as the embedding layers, produce an output of DimensionModel = 512.

Decoder layer. The decoder layer takes this encoded feature vector and creates the output token sequence. The decoder also has a set of equal levels n = 24. In addition to the two sublayers in each encoder layer, the decoder inserts a third sublayer that pays multiple attentions to the output of the encoder stack [130]. Like the encoder, it uses residual connections around each sublevel followed by level normalization. The self-focus sublayer in the decoder stack is also modified so that the position does not go into the subsequent position [5]. This masking, combined with fact that the outputed

embeddings are offset by one position, ensures that the predictions for the next position can only depend on the known outputs in the previous positions.

Attention mechanism. The disadvantage of the encoder-decoder structure is that the performance of the model degrades as the length of the original sequence increases due to the limitation on how much information an encoded feature vector of a fixed length can contain. To address this problem, Bahdanau, D. et al. [87] proposed an attention mechanism. In the attention mechanism the decoder tries to find the point in the encoder sequence where the most important information can be found and uses that information and previously decoded words to predict the next token in the sequence. A function of attention is to map a query and arrange a set of key value pairs in the output where query, keys, values and output are all vectors. The output is calculated by the weighted portion of the values, where the weight assigned to each key is correlated with the query with the function matched with that key [88].

5.1.2 AdamW Optimizer

The main contribution of AdamW [91] is to improve regularization in Adam [90] by separating weight reduction from gradient-based updating. In a comprehensive analysis, it is shown that Adam generalizes significantly better with split weight decomposition than with L2 regularization, achieving a 15% relative improvement in test error. This is true for various image recognition data sets (CIFAR-10 and ImageNet32x32), training budgets (100 to 1800 epochs) and learning rate graphs (fixed, stepped and cosine annealing) [91]. The weight reduction of AdamW optimizer makes the optimal settings for the learning rate and weight reduction factor much more independent, which simplifies hyperparameter optimization.

By disconnecting the weight loss and loss-based rating updates in Adam, this creates a variant of Adam (AdamW) with separate weight loss. After showing that L2 regularization and weight attenuation regularization are different in the adaptive gradient algorithm, the question arises how they differ and how their effects are interpreted. Equivalence with standard SGD is very intuitive. Both mechanisms bring the weight closer to zero at the same rate.

However, this is not the case with the adaptive gradient algorithm. The L2 regularization applies the sum of the loss function gradient and the regularization gradient (ie, the weight of the weight L2), but only the attenuation gradient of the

separated weight. The loss function is applied (using the weight decay step separated by the adjusted gradient procedure) [91]. In L2 regularization, both types of gradients are normalized by their characteristic (abbreviated) volume, so that the weight x with a larger characteristic gradient is greater than the other weights. In contrast, detached weight attenuation regulates all weights at the same rate λ and weights x with a weight greater than the standard L2 regularization.

According to Loshchilov et al. [91] weight decay is equal to scale-adjusted L2 regularization for adaptive gradient algorithm with fixed preconditioner. Let O denote an algorithm with fixed preconditioner matrix:

$$M_t = \text{diag}(s)^{-1} \text{ (with } s_i > 0 \text{ for all I)} \quad (7)$$

Then, O with base learning rate α executes the same steps on batch loss functions ft(θ) with weight decay λ as it executes without weight decay on the scale-adjusted regularized batch loss:

$$f_t^{sreg}(\theta) = f_t(\theta) + \frac{\lambda'}{2\alpha} \|\theta \odot \sqrt{s}\|_2^2 \quad (8)$$

where $\odot$ and $\sqrt{\cdot}$ denote element-wise multiplication and square root, and $\lambda' = \frac{\lambda}{\alpha}$.

It is pointed out that this theorem does not apply directly to practical adaptive gradation algorithms, as this changes the preceding position with each step. However, it can provide an intuition that the equivalent loss function is optimized at each step: parameters θ_i with a large inverse pre-conditioner s_i (which in practice would be caused by historically large gradients in dimension i) are regularized relatively more than they would be with L2 regularization, specifically, the regularization is proportional to $\sqrt{s_i}$ [91]. In XLCNN learning rate was set to $1 \cdot 10^{-3}$, AdamW's epsilon for numerical stability was set to $1 \cdot 10^{-6}$, decoupled weight decay was set to 0.0, while the bias should be corrected in each epoch.

5.1.3 Linear Schedule with Warmup

Linear schedule with warmup creates a schedule with a learning rate that decreases linearly from the initial learning rate set in the optimizer to 0. In the XLCNN model the AdamW optimization algorithm, which has been configured with a learning rate α and a warm-up factor ω has been implemented. The ω symbol is a sequence of “warm-up factors” where $\omega_t \in [0, 1]$, which are used to reduce the step size of each

iteration t. In particular, XLCNN implements a warm-up program by replacing α with $\alpha_t = \alpha \cdot \omega_t$ in the algorithm's update rule [92]. A linear warm-up configured by a "warm-up period" T was applied by XLCNN model :

$$\omega_t^{(linear,T)} = min\left(1, \frac{1}{T} \cdot t\right) \quad (9)$$

In order to calculate the value for the number of warmup steps, the following formula was used in XLCNN:

$$s_t = 0.25 \cdot \sum_{i=0}^{l_t} (i_t), \quad (10)$$

where s_n is the number of warmup steps, l_t is the total size of the training dataset and $t \in Z$ is the iteration on each step.

5.1.4 Embeddings

Given a sequential text input of T words $\{w_1, w_2, \ldots, w_T\}$, each word $w_t \in W$, is embedded into a d^{wrd}-dimensional vector space by using the following mathematical function:

$$\varphi_\theta(w_t) = Ef_t \quad (11)$$

The d^{wrd}-dimensional matrix E $\in \Re d^{(wrd x |W|)}$ represents all the learnable word embeddings in the current layer, just as the other parameters of the neural network. In particular, word embeddings use a lookup table with a much simpler array indexing operation, where $E \in \Re^{dwrd}$ represents the embedding of the word w_t. For each word in the sequence this lookup table function is then applied. The following matrix is produced from the concatenation from all resulting word embeddings.

$$\varphi_1^\theta(w_1, w_2, \ldots, w_T) = (E_{w1}, E_{w2}, \ldots, E_{wT}) \quad (12)$$

where $E_{wt} \in \Re^{(dwrd\, x\, T)}$ [93].

The usual learned linear transformation and softmax function to convert the decoder output to predicted next-token probabilities is also used in XLCNN model. XLCNN share the same weight matrix between the two embedding layers and the pre-softmax linear transformation, similar to [6]. Those weights were multiplied in the embedding layers, by $\sqrt{d_{model}}$.

5.1.5 Dropout

Dropout is a specific neural reduction strategy implemented in the XLCNN's feed forward neural network in order to significantly optimize its architecture while maintaining competitive performance. The result is the creation of a much smaller and

faster model due to the fact that some neurons have been zeroed. It also shows that the model was trained from the beginning without the use of a pre-trained model. Overall, the application of Dropout in a network of transformers offers the following key advantages [95]:

- Dropout deeply transforms transformers, stabilizes training and offers top performance at various benchmarks.
- Small and efficient models of any depth can be automatically extracted from larger models trained during the test without the need for modification.
- Dropout is easy to apply in any Transformer model.

The application of Dropout in XLCNN has as a result the prevention of excessive overfitting and provides a way to effectively integrate multiple architectural neural networks. The term "dropout" refers to those who enter the neural network (hidden and visible) [94]. Disconnection from the unit results in the temporary termination of the network with its incoming and outgoing connections. In the simplest case, each unit is stored independently of the other units with a fixed probability p, where p can be selected from the test set [94]. In XLCNN the probability p was set to 0.1, which seems according to the results described in Chapter V, to be close to optimal for the feed forward neural networks.

The XLCNN model includes all units that survived the dropout. A neural network with n units can be thought of as a set of 2n possible thin neural networks. All these networks have the same weighting, so the total number of parameters is still $O(n^2)$ or less. For each presentation of each training example, a new thin network is selected and trained. Thus, training a weighted neural network can be thought of as training a 2n thin network with wide load sharing, where each thin network is trained very rarely, if at all.

5.1.6 Activation function

In XLCNN model it was proposed the Gaussian Error Linear Unit (GELU), a neural network activation function with high performance. $x\Phi(x)$, is the GELU activation function where $\Phi(x)$ is the standard Gaussian cumulative distribution function [16]. XLCNN model used an “adaptive dropout network” which was trained jointly with the GELU activation function by approximately computing local expectations of binary dropout variables, computing derivatives using back-propagation, and using stochastic gradient descent [97].

GELU activation function is a combination from ReLUs [98], zoneout [131] and dropout [97]. First, we notice that both ReLU activation function and dropout produce a neuronal output. The ReLU deterministically multiplies the input by zero or one, and the dropout stochastically multiplies the input by zero. In addition, a new regularizer called zoneout multiplies the inputs by one [131]. This functionality was merged by multiplying the input by zero or one, but the values of this zero-one mask are stochastically determined depending on the input [16].

In particular, the neuron input x can be multiplied by m ~ Bernoulli (Φ(x)), where Φ(x) = P (X ≤ x), X ~ N (0, 1) is the cumulative distribution function of the standard normal distribution. This distribution is chosen because of the fact that neuron inputs tend to follow a normal distribution especially with Batch Normalization. In this setting, as x decreases, the inputs have a higher probability of being “dropped”, so the transformation applied to x is stochastic and depends upon the input. This masking of the input preserves non-determinism and the dependence of the input value as well. The stochastically chosen mask is reduced to a stochastic zero or an identical transformation of the input data.

Because the desired result from a neural network must be a deterministic decision, the above assumption leads to nonlinearity. The nonlinearity is the expected change of the stochastic regularizer at the input x, which is Φ (x) × Ix + (1 - Φ (x)) × 0x = xΦ (x). This expression implies that x can be measured by how much larger it is than other inputs. The cumulative distribution function of Gaussian is always treated with the error function, the Gaussian error linear component (GELU) can be defined as [16]:

$$GELU(x) = xP(X \le x) = x\Phi(x) = x \cdot \frac{1}{2}\left[1 + erf\left(\frac{x}{\sqrt{2}}\right)\right] \qquad (13)$$

where x is the input matrix. GELU can be defined approximately with:

$$0.5x\left(1 + \tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right) \qquad (14)$$

or

$$x\sigma(1.702x) \qquad (15)$$

if greater feedforward speed is worth the cost of exactness [16].

5.2 The Proposed XLCNN Model

The proposed XLCNN is a Transformer model having an encoder-decoder structure. The main purpose of the encoding layer is to achieve a mapping between a sequence of input symbol representations (x_1, ..., x_n) and a sequence of continuous representations z = (z_1, ..., z_n). The decoder having the value of the representation z, outputs a sequence (y_1, ..., y_m) of symbols one element at a time [6]. At each step, the model is spontaneously accepting the input of the symbols created in the previous step to create the next symbols. XLCNN's model architecture is a multilayer bidirectional Transformer model based on the original implementation described in Yang, Z et al. [6]. XLCNN uses a composite self-attention layer and two fully connected layers inside the feed forward network, as shown on the Figure 12.

XLCNN uses embeddings [86] with $32 \cdot 10^3$ token vocabulary. The first token of each sequence is always a special classification token ([CLS]). The last hidden state corresponding to this token is used as a total sequence representation for classification functions. Pairs of sentences are packed in the same sequence. The sentences were separated in two ways. Firstly, they are separated with a special token ([SEP]) [9]. Secondly, a learned embedding was added to each token that indicates whether it belongs to the benign set or the malware set. For a particular token, its input representation is built by combining the corresponding categories and position embeddings.

XLCNN uses convolutional neural networks (CNNs) [9] in its core architecture. Specifically, it uses 24 layers of feed forward neural networks, with each layer consisting of 2 CNNs having as input a matrix of dimensions [512 x 1 x 512] and producing an output matrix of [512 x 1 x 2048] dimensions, 2 Linear Transformation layers and 1 Normalization layer. Our experimental results have shown that this neuron architecture is the most efficient in malware classification, as it increases network performance, reduces loss [100] and increases ultimate prediction accuracy.

The main benefit of this approach is that it allows the network to be learning while remaining unchanged, instead of having to be coded in the network as is the case with the XLNet model. Our feed forward architecture with CNNs can learn much more complex invariants (e.g to recognize malicious code only from the metadata it has), while maintaining many of the advantages of the low number of CNN parameters [101]. This results in faster training and easier learning compared to the XLNet. Figure 12 shows the architecture of feed forward convolutional neural network.

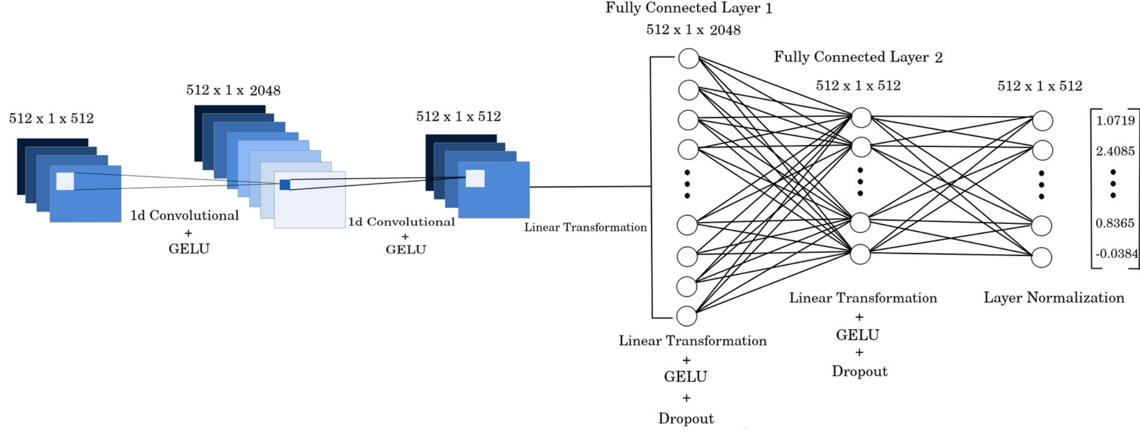


Figure 12: Feed forward convolutional neural network architecture with input matrix [512 x 1 x 512] and output matrix [512 x 1 x 2048].

The vocabulary size of the XLCNN model, which defines the number of different tokens that can be represented by the different input words, is set to $32 \cdot 10^3$. The linear transformation is the same in different positions, but uses different parameters for each layer. The additional convolutions neural networks have kernel size of 1. The inner-layer has dimensionality $n_x = 512$ and the dimensionality of output is $n_{fS}= 2048$. We proposed 24 hidden layers in the transformer encoder and 16 attention heads for each attention layer.

The feed forward network of XLCNN model consists of intertwined layers of self-focus and direct communication. Each feed-forward layer is a positional function that processes each input vector independently. Let $X \in R^d$ be a vector corresponding to some input text prefix. The feedforward layer FF(·) can be expressed as follows (biased terms omitted):

$$FF(x) = f(x \cdot K^T) \cdot V \tag{16}$$

where, f is a non-linear function such as GeLU and K, $V \in R^{dm \times d}$ are parameter matrices [5].

The effectiveness of XLCNN's feed forward network depends on the memory of previous decisions. Neural memory [89] consists of d_m key-value pairs that we call memories. Each key is represented by a d-dimensional vector $k_i \in R^d$, and together they form a matrix of parameters $K \in R^{dm \times d}$. Analogously, the value parameter is defined as $V \in R^{dm \times d}$. Given the input vector $x \in R^d$, the distribution can be calculated by the

entire key and uses it to calculate the expected value: $p\left(k_i \middle| x\right) \propto \exp\left(x \cdot k_i\right)$ (11)

$$MN\left(x\right)=\sum_{i}^{d_m} p\left(k_i \middle| x\right) v_i \tag{17}$$

where MN(x) is the neural memory of feed forward neural network. When the matrix is added, a more compact formulation is created [89]:

$$MN\left(x\right)=softmax\left(x \cdot k_T\right) \cdot V \tag{18}$$

In addition to the attention sublevels, each of the encoder and decoder layers contains a fully connected anticipation network that is applied to each position separately and identically. It consists of two linear transformations with GeLU [16] activation in between. The new feed forward neural network equation is as follows:

$$FFN\left(x\right)=max\left(0,xW_1+b_1\right) \cdot W_2+b_2 \tag{19}$$

where $x \in R^d$ is the input vector, W_1, W_2 the weights and b_1, b_2 the biases. The Figure 13 shows the graphical representation of the fully connected feed forward neural network proposed for XLCNN.

The feed forward network accepts as input a vector of dimensions [512 x 1 x 512], which is multiplied by the learnable weight of dimensions [512 x 1 x 2048]. The [512 x 1 x 2048] dimension matrix is multiplied by a constant α value and the result of the multiplication between the learnable bias and the beta value is added. This convolutional equation is represented as follows:

$$C\left(x\right)=\beta \cdot b_m+a \cdot \left(x \cdot w_m\right) \tag{20}$$

where β and α were set to 1, b_m is the bias, $x \in R^d$ is the input vector and w_m is the learnable weight.

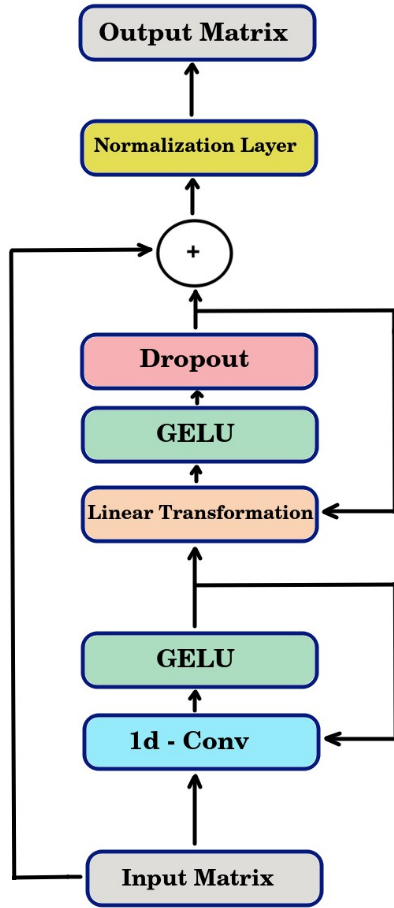


Figure 13: XLCNN feed forward neural network structure.

In the next step the error of the above function was calculated using the Gaussian error function [132] which is given by the completion of the Euler error [133] elevated to -x in the square with limits from -x to x. The effect of completion multiplied by 2 for the root π gives the possibility that a normally distributed random variable having an average of 0 and a variation of 0.5 falls into the region [−x, x]. The Gaussian error function can be computed:

$$erf(x) = \frac{2}{\sqrt{\pi}} \int_{0}^{x} e^{-x^2} dx \tag{21}$$

where x is the result of the convolutional equation. Euler's equation was used to calculate the Median Absolute Deviation (MAD) [134] through the equation:

$$MAD(x)=\sigma \cdot \sqrt{2} \cdot erf(x)^{-1} \cdot \frac{1}{2} \quad (22)$$

where σ=1 is the standard deviation, erf(x) is the Gaussian error function and x is the input vector.

The Cumulative Distribution Function (CDF) [135] was then calculated, which shows the probability that a value of the function is between zero and a value x, as shown in the following mathematical form:

$$G(x)=MAD(x) \cdot \int_0^x e^{\frac{-(x-\mu)^2}{2 \cdot \sigma^2}} dx \quad (23)$$

where the parameter μ is the mean or expectation of the distribution, σ is the standard deviation and x is the input vector. The result of function G(x) was used as input for the convolutional network, but also used as input for function G(x). The result of the second iteration marks the end of the first sub-infrastructure and the start of the second as described below.

Initially, the linear transformation function accepts as input the result of G(x) whose dimensions are [512 x 1 x 512] and produces a dimension matrix [512 x 1 x 2048]. It is essentially a mapping between the value G(X) and the weights that are towards the learning process [136]. The mathematical representation of the above sentence is described as:

$$y=G(x) \cdot w^T+b \quad (24)$$

where w is the learnable weight of the function y, b the biases and G(x) is the CDF.
After the application of the GELU function which takes as input the result of the y function, some elements of the matrix with probability p=0.1 are zeroed according to the definition of the Bernouli distribution [137]. Those tensors that remained intact and did not zero were multiplied by a factor equal to $\frac{1}{1-p}$. The Bernouli distribution applied as:

$$f(x, p) = \begin{cases} x, if\ x \geq p \\ 0, if\ x < p \end{cases} \quad (25)$$

In the second iteration of the second sub-infrastructure the result of the function f(x, p) was a matrix of size [512 x 1 x 512] and was added with the tensors that were

used as input at beginning before introduced into the feed forward network. In order to reduce the training time of the feed forward neural network but also to reduce the sensitivity in the final state, the normalization function [138] was introduced which is given by the following formula:

$$y = \frac{x - E[x]}{\sqrt{Var[x] + \varepsilon}} \cdot \gamma + \beta \qquad (26)$$

where $\varepsilon = 10^{-5}$ is the denominator for numerical stability, γ and β are learnable affine transform parameters and x is the input matrix. After this step the output matrix size remained [512 x 1 x 512], as shown in Figure 12.

This repetitive structure but also the architecture itself makes the XLCNN unique in its kind. The purpose of the repeating structure is to develop more complex connections between neurons without having to increase the size of the matrix used as input, thus reducing the execution time of mathematical operations.

5.3 Application on malware detection

A state of the art analysis was used to identify the malware which results from the combination of XLCNN with the metadata of the Windows executable files. Different sources were used to create the data set and combined for the purpose of model training. To create the vectors, a tokenizer [96] was used which is specially trained for the purpose of detecting malware. Figure 14 shows the steps followed for classifying files.

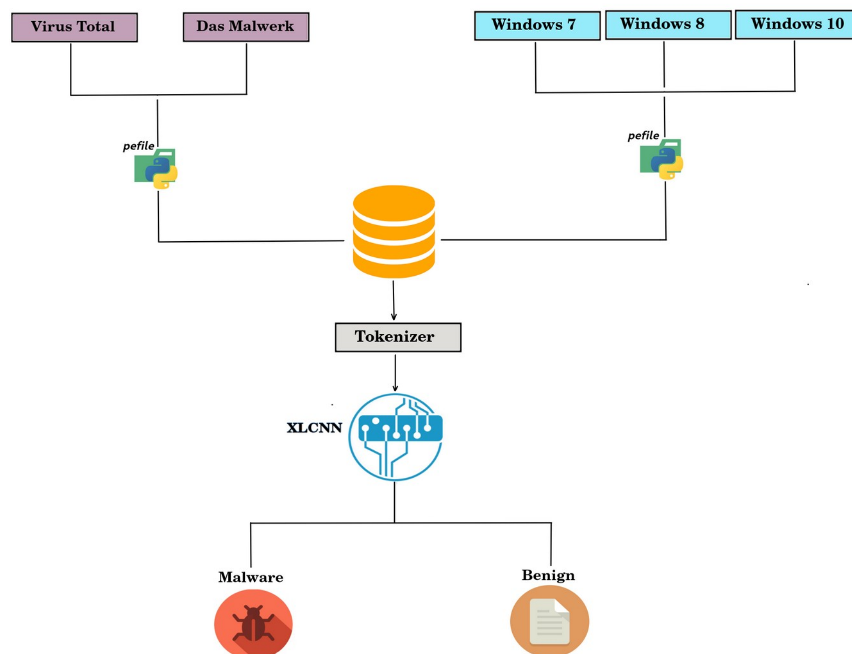


Figure 14: Process of classifying executable files into malicious and benign.

Initially, executable files consisting of 2 categories were collected, the infected files and the benign ones. Malicious archives come from two different sources, the Virus Total [106] and the Das Malwerk [107]. The benign files were collected from the 3 distributions of Windows 7, 8 and 10 [108], [109], [110], which did not contain infected files, as they came from clean installations. A python script file was then created and the *pefile* library [104] was used to export metadata from all executable files collected, whether they were malware or benign. They were then stored in a csv file and divided into training data and testing data. The metadata became vectors through the tokenizer as described by Sennrich R. et al. [105] and was used as input for the XLCNN model. After the training process the model recognized the hostile executable files with a success rate of 95.07%. The following sections present in detail the steps followed for dataset creation, metadata extraction and the results of XLCNN.

5.3.1 Source collection for malware detection

Two different data sets for malicious executable files were used and created from scratch. The first one was created with the help of VirusTotal. VirusTotal is one of the most popular and widely used scanning services by researchers and industry professionals [106]. VirusTotal provides file scanning services (for malware analysis) and URL scanning services (for detecting malware and phishing hosts). It works with more than 70 security vendors to aggregate the results of their analyses [111].

The second set of data was retrieved from Das Malwerk [107] which has various types of malware on its website. To retrieve the files, a python script was created from scratch using its *request* [115] and *beautifulsoup* libraries [116].

The total number of malicious files retrieved from VirusTotal and Das Malwerk was 3,117, with no duplicates, while the total number of benign data retrieved from the three Windows distributions was 1,794, also without duplicates. The combination of the total data constitutes the final number 4911. Of these, 10% of the malicious data and 10% of the benign data were used to verify the XLCNN model. Therefore, 4452 size of data were used for training and 446 size of data for verification. Within 100 epochs of training, XLCNN managed to achieve a 95.07% accuracy rate.

5.3.2 Metadata extraction

With the growing availability of malware resources the quality of the learning process depends on the availability of these resources as well as the amount and expression of metadata used to explain them. The availability of malware resources is

largely provided by storing these objects or their metadata in digital repositories. Reserves generally provide malware resources for exploration, demonstration and acquisition. Metadata is an important factor in the ability to find learning objects as information provides additional details of learning objects. These descriptions may relate to memory calls, address allocations, product name, product version, entropy and hash signature.

Metadata were divided into two categories [102]. The first category consists of metadata that describes the properties of the object that are not related to the domain to which the object belongs. This metadata is general and can be applied to all learning objects, regardless of domain or discipline. Examples of such metadata are file format, language and so on.

The second category pertains to metadata that describes learning objects with domain-specific information. Many domains have developed classifications that classify content within a particular domain. As an example of a domain-specific metadata is the description of memory allocations in executable files in the field of cyber security.

Note that no feature export algorithm was used, but all available metadata was used as input for XLCNN model. The metadata in this research project was extracted from the executable Windows files using a python library called *pefile* [104]. *Pefile* is a multi-platform Python module to parse and work with Portable Executable (PE) files [103]. Most of the information contained in the PE file headers is accessible as well as all the sections details and data. The structures defined in the Windows header files are accessible as attributes in the PE instance. The naming of fields/attributes tries to adhere to the naming scheme in those headers. Some of the tasks that Pefile library makes possible are [104]:

- Inspecting headers
- Analyzing of sections' data
- Retrieving embedded data
- Reading strings from the resources
- Warnings for suspicious and malformed values

5.4 Experimental implementation and results

The following experiments are done in order to evaluate the performance of the proposed XLCNN. The experiments mainly examine the effect of feed forward convolutional neural networks which accept 512 size tensors as input and produce 2048 size tensors as output. As shown in Figure 15 one can observe that sensitivity is a fairly large fact which suggests that the percentage of false negatives is small. From Table 6 we

observe that out of the total of 283 malwares, XLCNN successfully recognized 277 of them while the 6 were erroneously estimated that they are benign files. Respectively, out of the 163 benign files it successfully recognized 147 while 16 were erroneously estimated that they were malicious files.

Table 6. The results for sensitivity and specificity for XLCNN transformer model

Desicion	XLCNN Transformer	
	Hypothesis True	**Hypothesis False**
Malware	277	16
Benign	147	6

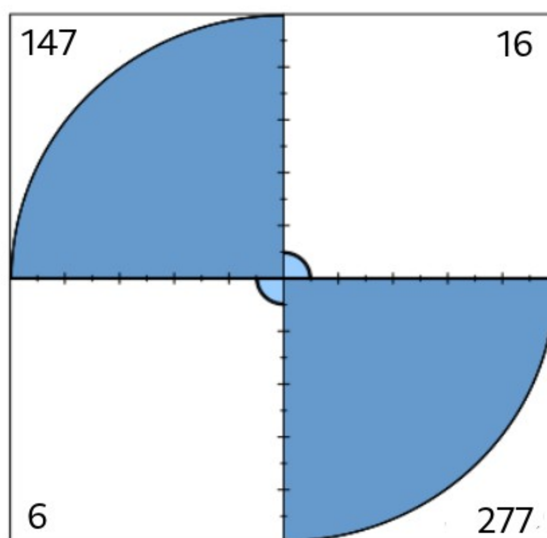


Figure 15: Confusion matrix for test set.

At FPR 9.82%, we reported the recall or true positive rate at 97.88%, the precision or positive predictive value rate at 94.54%, F1 score at 96.18%, AUC at 94.03%, accuracy at 95.07%. Miss rate or false negative rate (FNR) calculated at 9.82%, specificity or true negative rate (TNR) at 90.18%, miss rate or false negative rate (FNR) at 2.12%, negative predictive value (NPV) at 96.08% and false discovery rate (FDR) at 5.46% as shown on Table 7.

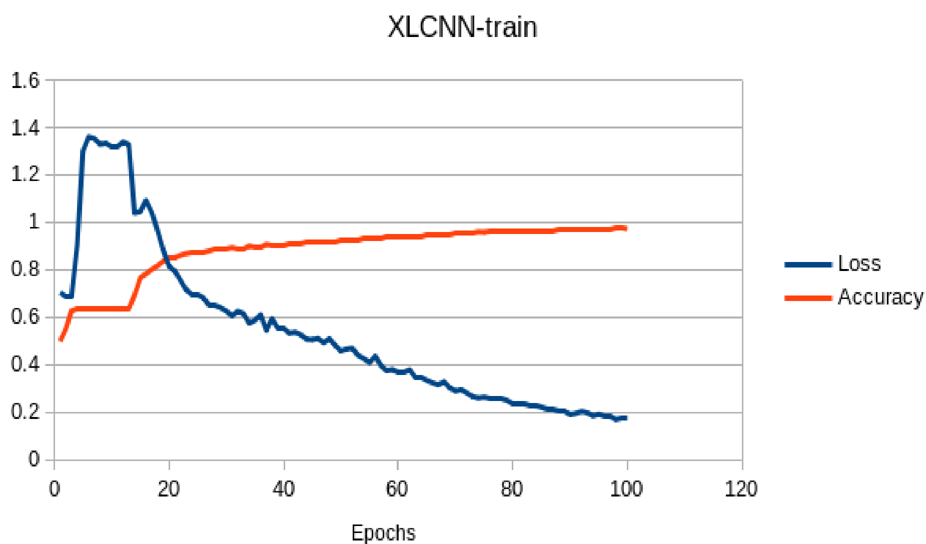


Figure 16: XLCNN training process.

Figure 16 shows the process of training the XLCNN for a period of 100 epochs while Figure 17 shows the process during which the neural network is verified on data which was not trained to determine the progress of the model. As shown in Figure 16 and Figure 17 the model has a linear increase in performance until the 3rd epoch while from the 4th to the 13th there is no further increase, but the performance remains constant. From the 14th epoch until the last a logarithmic increase is observed. Note that during the fine-tune process, our model stores those weights that have the highest performance during the validation process regardless of how large the percentage is during training. XLCNN took just 69 seasons to achieve maximum accuracy of 95.07%. The red line represents XLCNN's accuracy and the blue line represents the loss.

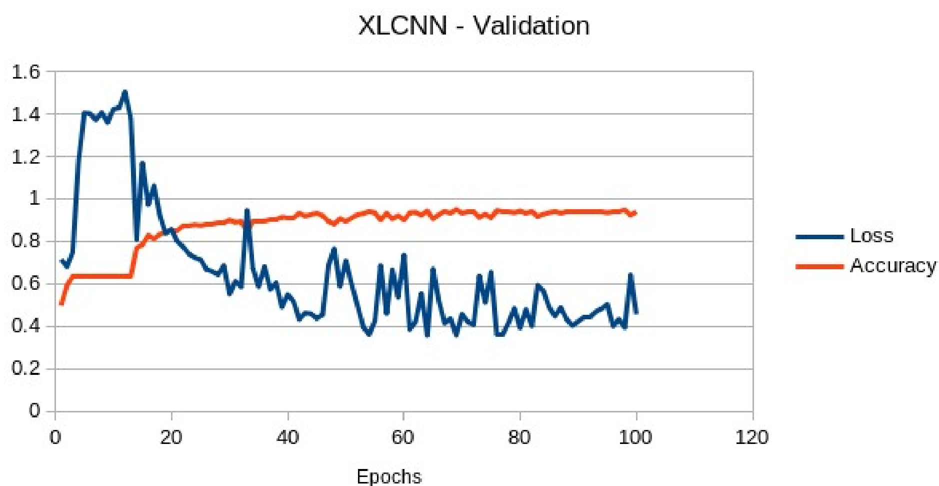


Figure 17: XLCNN testing process.

Figure 18 shows the training of XLCNN this time not depending on the epochs, but depending on the repetitions per epoch. Virtually every epoch consists of 4452 repetitions. In total the repetitions after 100 epochs are 445200. This diagram shows more clearly the fluctuations of effectiveness during the training.

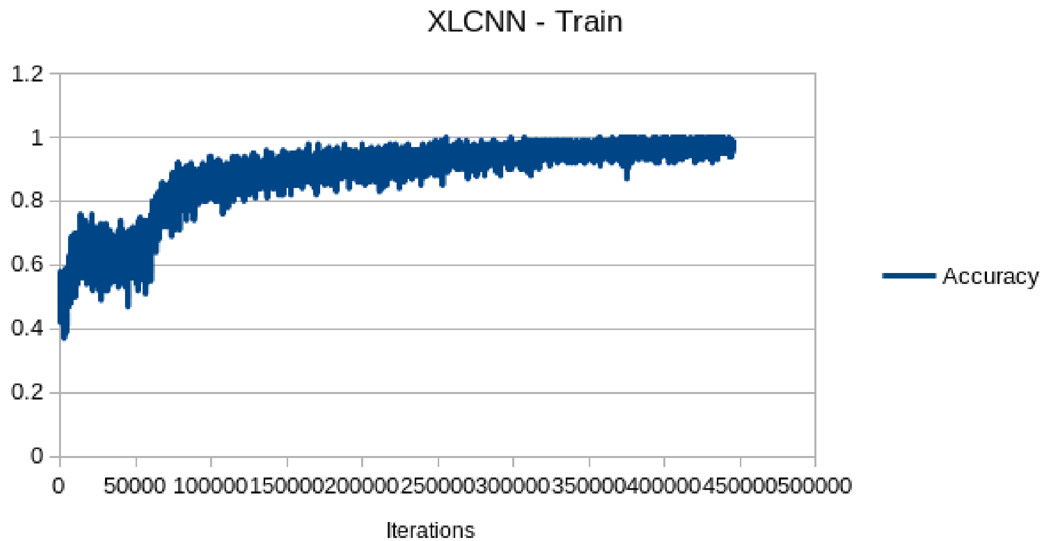


Figure 18: XLCNN training process per iteration.

5.4.1 Comparison with XLNet

After training for 100 epochs, XLCNN managed to achieve a score of 95.07% in the testing data set and surpassed the XLNet which achieved a score of 93.49% as shown in Table 7. We observe the recall of XLCNN is 97.88% clearly higher than that of XLNet with a percentage of 96.82%. The false positive ratio (FPR) of XLCNN is lower by 2.45% while the True Negative Rate (TNR) is a higher, fact that proves that it predicts malicious files with greater success compared to XLNet. The false negative rate (FNR) of XLCNN is 1.5% lower and therefore less likely to incorrectly predict a malicious file as benign. XLCNN predicts malware files with 94.54% rate and benign files with 96.08% rate compared to XLNet which has a precision rate of 93.19% and negative predictive value (NPV) rate of 94.08%. The mistaken rejection of an actually malicious file (FDR) of XLCNN is 1.34% lower compared with XLNet. Finally as we can see XLCNN has f1 score of 96.18% and AUC rate of 94.03%, higher than those of XLNet which has f1 score of 95.97% and 92.27% AUC rate.

Table 7. Comparison with state-of-the-art results on the test set between XLCNN and XLNet

	recall	**FPR**	**TNR**	**FNR**	**precision**	**NPV**	**FDR**	**f1**	**AUC**	**accuracy**
XLCNN	97.88	9.82	90.18	2.12	94.54	96.08	5.46	96.18	94.03	95.07
XLNet	96.82	12.27	87.73	3.18	93.19	94.08	6.80	94.97	92.27	93.49

Note that both the size of the data used as input and the size of the neural networks were kept the same in both models, so that the final measurement is meritocratic and the effectiveness of one model over the other is decided. This proves that the feed forward neural network based on convolutional neural networks is more efficient than the classic XLNet, while the training time remains the same.

6 Epilogue

In the present thesis, the techniques with which the analysis of malware is achieved were initially mentioned. Typically, two methods were mentioned in order to analyze malware. Static and dynamic analysis. Some categories of malware were then reported, as well as the way in which they can infect a system or an organization, and also the methods they use to avoid detection by antivirus software. Knowing the types of malware and the techniques they use both to attack and to avoid detection, a state of the art algorithm was developed based on the transformer models called XLCNN. A transformer is a deep learning model that adopts the mechanism of attention, differentially weighing the significance of each part of the input data. It is used primarily in the field of natural language processing (NLP), but in this work it was used to classify executable files into malicious and bening. In order to achieve such a thing, the attention mechanisms were studied, which are used for sequence modeling. Finally, the architecture of XLCNN was explained and compared to the XLNet model, which was created by the Google team.

6.1 Summary and conclusions

In the present thesis, the creation of the state-of-the-art transformer model XLCNN proved that the addition of CNNs to the feedforward network in a combination with Linear Transformation, GELU, Dropout and the Normalization layer offer a higher success rate of 95.07% compared to the XLNet model which had a success rate of 93.49%. The same sizes of neural networks were used both in the architecture part and in the data part, so that there is a meritocratic comparison of the two models. It was decided that XLCNN was more accurate and we observe after fine-tune procedure, that out of the total of 283 malwares, XLCNN successfully recognized 277 of them, while the 6 erroneously estimated that they are benign files. Respectively, XLCNN recognized 147 out of the 163 benign files, while 16 erroneously estimated that they were malicious files. The analysis of the executable files was based on the metadata they have, which makes malware detection quite effective in combination with the addition of the trained XLCNN Transofrmer model.

6.2 Limits and limitations of research

The model according to the results obtained could achieve an even greater success rate in its prediction, if it had more data for its training. The success of the XLCNN was determined by its effective classification, while it was trained with a small amount of data, limitation that we cannot ignore. Another restriction was the size of the tensors used as input. This limitation has to do with the memory of the graphics card used to calculate the mathematical operations. The input size was [512 x 1 x 512] and the output size was [512 x 1 x 2048], i.e. the maximum allowed by the computer. Possibly, if larger matrix sizes had been used as input, the neural network could stand out more characteristics to make the classification. Despite the limitations described above, the experimental results have shown that XLCNN is on of the most effective model for malware detection in terms of Windows executable files compared to those described in related work section.

6.3 Future Extensions

The deep learning technique used belongs to the category of supervised learning. A future expansion could be the creation of a network to detect malware without supervision. A deep reinforcement learning algorithm like deep q learning [117] or actor critic reinforcement learning [118] could be applied, which would take the XLCNN as a pre-trained model.

References

[1] Jha, S., Prashar, D., Long, H. V., & Taniar, D. (2020). Recurrent neural network for detecting malware. *Computers & Security, 99*, 102037.

[2] Lipton, Z. C., Berkowitz, J., & Elkan, C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning.

[3] Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, *9*(8), 1735-1780.

[4] Chung, J., Gulcehre, C., Cho, K., & Bengio, Y. (2014). Empirical evaluation of gated recurrent neural networks on sequence modeling. In *NIPS 2014 Workshop on Deep Learning, December 2014*.

[5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., ... & Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems* (pp. 5998-6008).

[6] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., & Le, Q. V. (2019). XLNet: Generalized Autoregressive Pretraining for Language Understanding. *Advances in Neural Information Processing Systems*, *32*, 5753-5763.

[7] Povey, D., Cheng, G., Wang, Y., Li, K., Xu, H., Yarmohammadi, M., & Khudanpur, S. (2018, September). Semi-Orthogonal Low-Rank Matrix Factorization for Deep Neural Networks. In *Interspeech* (pp. 3743-3747).

[8] Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., & Dean, J. (2017). Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.

[9] Wu, J. (2017). Introduction to convolutional neural networks. *National Key Lab for Novel Software Technology. Nanjing University. China*, *5*(23), 495.

[10] Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019, January). BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*.

[11] Kaiser, Ł., & Bengio, S. (2016). Can Active Memory Replace Attention?. *Advances in Neural Information Processing Systems*, *29*, 3781-3789.

[12] Kalchbrenner, N., Espeholt, L., Simonyan, K., Oord, A. V. D., Graves, A., & Kavukcuoglu, K. (2016). Neural machine translation in linear time. *arXiv preprint arXiv:1610.10099*.

[13] Gehring, J., Auli, M., Grangier, D., Yarats, D., & Dauphin, Y. N. (2017, July).

Convolutional sequence to sequence learning. In *International Conference on Machine Learning* (pp. 1243-1252). PMLR.

[14] Xiong, R., Yang, Y., He, D., Zheng, K., Zheng, S., Xing, C. & Liu, T. (2020, November). On layer normalization in the transformer architecture. In *International Conference on Machine Learning* (pp. 10524-10533). PMLR.

[15] Britz, D., Goldie, A., Luong, M. T., & Le, Q. (2017, September). Massive Exploration of Neural Machine Translation Architectures. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing* (pp. 1442-1451).

[16] Hendrycks, D., & Gimpel, K. (2016). Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415*.

[17] Press, O., & Wolf, L. (2017, April). Using the Output Embedding to Improve Language Models. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers* (pp. 157-163).

[18] Dai, A. M., & Le, Q. V. (2015). Semi-supervised sequence learning. *Advances in neural information processing systems*, *28*, 3079-3087.

[19] Peters, M. E., Neumann, M., Iyyer, M., Gardner, M., Clark, C., Lee, K., & Zettlemoyer, L. (2018). Deep contextualized word representations. In *Proceedings of NAACL-HLT* (pp. 2227-2237).

[20] McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in Translation: Contextualized Word Vectors. *Advances in Neural Information Processing Systems*, *30*.

[21] Gong, X. R., Jin, J. X., & Zhang, T. (2019, November). Sentiment analysis using autoregressive language modeling and broad learning system. In *2019 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (pp. 1130-1134). IEEE.

[22] Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q., & Salakhutdinov, R. (2019, July). Transformer-XL: Attentive Language Models beyond a Fixed-Length Context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics* (pp. 2978-2988).

[23] Aman, W. (2014). A framework for analysis and comparison of dynamic malware analysis tools. *arXiv preprint arXiv:1410.2131*.

[24] Moser, A., Kruegel, C., & Kirda, E. (2007, May). Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP'07)* (pp. 231-

245). IEEE.

[25] Zhuge, J., Holz, T., Song, C., Guo, J., Han, X., & Zou, W. (2009). Studying malicious websites and the underground economy on the Chinese web. In *Managing information risk and the economics of security* (pp. 225-244). Springer, Boston, MA.

[26] Kushner, D. (2013). The real story of stuxnet. IEE *Spectrum, 50*(3), 48-53.

[27] Porras, P., Saidi, H., & Yegneswaran, V. (2009). An analysis of conficker's logic and rendezvous points. *Computer Science Laboratory, SRI International, Tech. Rep*, 36.

[28] Chen, H., Dean, D., & Wagner, D. A. (2004, February). Model Checking One Million Lines of C Code. In *NDSS* (Vol. 4, pp. 171-185).

[29] Feng, H. H., Giffin, J. T., Huang, Y., Jha, S., Lee, W., & Miller, B. P. (2004, May). Formalizing sensitivity in static analysis for intrusion detection. In *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004* (pp. 194-208). IEEE.

[30] Christodorescu, M., and S. Jha. *Static analysis of executables to detect malicious patterns. WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES*. 2006.

[31] Moser, A., Kruegel, C., & Kirda, E. (2007, December). Limits of static analysis for malware detection. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)* (pp. 421-430). IEEE.

[32] Ye, Y., Li, T., Adjeroh, D., & Iyengar, S. S. (2017). A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, *50*(3), 1-40.

[33] Gandotra, E., Bansal, D., & Sofat, S. (2014). Malware analysis and classification: A survey. *Journal of Information Security, 2014*.

[34] Ucci, D., Aniello, L., & Baldoni, R. (2019). Survey of machine learning techniques for malware analysis. *Computers & Security*, *81*, 123-147.

[35] Abou-Assaleh, T., Cercone, N., Keselj, V., & Sweidan, R. (2004, September). N-gram-based detection of new malicious code. In *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* (Vol. 2, pp. 41-42). IEEE.

[36] Kirat, D., & Vigna, G. (2015). Malgene: Automatic Extraction of Malware Analysis Evasion Signature. CCS'15 Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, Colorado, USA, 12-16 October, 769-780.

[37] Hashemi, H., & Hamzeh, A. (2019). Visual malware detection using local malicious

pattern. *Journal of Computer Virology and Hacking Techniques*, *15*(1), 1-14.

[38] Shaid, S. Z. M., & Maarof, M. A. (2014). Malware behaviour visualization. *Jurnal Teknologi*, *70*(5).

[39] Salehi, Z., Sami, A., & Ghiasi, M. (2014). Using feature generation from API calls for malware detection. *Computer Fraud & Security*, *2014*(9), 9-18.

[40] Han, K. S., Kim, I. K., & Im, E. G. (2012). Malware classification methods using API sequence characteristics. In *Proceedings of the International Conference on IT Convergence and Security 2011* (pp. 613-626). Springer, Dordrecht.

[41] Cheng, Y., Fan, W., Huang, W., & An, J. (2017, September). A shellcode detection method based on full native api sequence and support vector machine. In *IOP Conference Series: Materials Science and Engineering* (Vol. 242, No. 1, p. 012124). IOP Publishing.

[42] Sihwail, R., Omar, K., & Ariffin, K. A. Z. (2018). A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis. *International Journal on Advanced Science, Engineering and Information Technology*, *8*(4-2), 1662.

[43] You, I., & Yim, K. (2010, November). Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications* (pp. 297-300). IEEE.

[44] Wong, W., & Stamp, M. (2006). Hunting for metamorphic engines. *Journal in Computer Virology*, *2*(3), 211-229.

[45] Ball, T. (1999, September). The concept of dynamic analysis. In *Software Engineering—ESEC/FSE'99* (pp. 216-234). Springer, Berlin, Heidelberg.

[46] Or-Meir, O., Nissim, N., Elovici, Y., & Rokach, L. (2019). Dynamic malware analysis in the modern era—A state of the art survey. *ACM Computing Surveys (CSUR)*, *52*(5), 1-48.

[47] Kang, M. G., Poosankam, P., & Yin, H. (2007, November). Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM workshop on Recurring malcode* (pp. 46-53).

[48] Royal, P., Halpin, M., Dagon, D., Edmonds, R., & Lee, W. (2006, December). Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)* (pp. 289-300). IEEE.

[49] Liang, G., Pang, J., & Dai, C. (2016). A behavior-based malware variant classification technique. *International Journal of Information and Education Technology, 6*(4), 291.

[50] Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, *44*(2), 1-42.

[51] Nebbett, G. (2000). *Windows NT/2000 native API reference*. Sams Publishing.

[52] Stallman, R. (2003). Free software foundation (FSF). In *Encyclopedia of Computer Science* (pp. 732-733).

[53] Hunt, G., & Brubacher, D. (1999, July). Detours: Binaryinterception ofwin 3 2 functions. In *3rd Usenix Windows Nt Symposium*.

[54] Hu, W., Mu, D., Oberg, J., Mao, B., Tiwari, M., Sherwood, T., & Kastner, R. (2014). Gate-level information flow tracking for security lattices. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, *20*(1), 1-25.

[55] Penard, W., & van Werkhoven, T. (2008). *On the secure hash algorithm family. National Security Agency*. Technical Report.

[56] Windows, Inside. (2008). From the February 2002 issue of MSDN Magazine.

[57] Chenke, L., Feng, Y., Qiyuan, G., Jiateng, Y., & Jian, X. (2017, July). Anti-Reverse-Engineering Tool of Executable Files on the Windows Platform. In *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)* (Vol. 1, pp. 797-800). IEEE.

[58] Zhang, K., Li, C., Wang, Y., Zhu, X., & Wang, H. (2017). Collaborative support vector machine for malware detection. *Procedia Computer Science*, *108*, 1682-1691.

[59] Baldini, G., & Geneiatakis, D. (2019, April). A performance evaluation on distance measures in KNN for mobile malware detection. In *2019 6th international conference on control, decision and information technologies (CoDIT)* (pp. 193-198). IEEE.

[60] Morales-Molina, C. D., Santamaria-Guerrero, D., Sanchez-Perez, G., Perez-Meana, H., & Hernandez-Suarez, A. (2018, November). Methodology for malware classification using a random forest classifier. In *2018 IEEE International Autumn Meeting on Power, Electronics and Computing (ROPEC)* (pp. 1-6). IEEE.

[61] Yin, C., Zhu, Y., Fei, J., & He, X. (2017). A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access*, *5*, 21954-21961.

[62] Barrett, T. K., & Sandler, D. G. (1993). Artificial neural network for the determination of

Hubble Space Telescope aberration from stellar images. *Applied Optics*, *32*(10), 1720-1727.

[63] Chen, Y., Jiang, H., Li, C., Jia, X., & Ghamisi, P. (2016). Deep feature extraction and classification of hyperspectral images based on convolutional neural networks. *IEEE Transactions on Geoscience and Remote Sensing*, *54*(10), 6232-6251.

[64] Zargar, S. A (2021). Introduction to Convolutional Neural Networks. Department of Mechanical and Aerospace Engineering, North Carolina State University.

[65] Jha, S., Prashar, D., Long, H. V., & Taniar, D. (2020). Recurrent neural network for detecting malware. *computers & security*, *99*, 102037.

[66] Rahali, A., & Akhloufi, M. A. (2021). MalBERT: Using Transformers for Cybersecurity and Malicious Software Detection. *arXiv preprint arXiv:2103.03806*.

[67] Hardy, W., Chen, L., Hou, S., Ye, Y., & Li, X. (2016). DL4MD: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Science (ICDATA)* (p. 61). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

[68] Rhode, M., Burnap, P., & Jones, K. (2018). Early-stage malware prediction using recurrent neural networks. *computers & security*, *77*, 578-594.

[69] Pascanu, R., Stokes, J. W., Sanossian, H., Marinescu, M., & Thomas, A. (2015, April). Malware classification with recurrent networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 1916-1920). IEEE.

[70] Idika, N., & Mathur, A. P. (2007). A survey of malware detection techniques. *Purdue University*.

[71] Namanya, A. P., Cullen, A., Awan, I. U., & Disso, J. P. (2018, August). The world of malware: An overview. In *2018 IEEE 6th International Conference on Future Internet of Things and Cloud (FiCloud)* (pp. 420-427). IEEE.

[72] Perdisci, R., Lanzi, A., & Lee, W. (2008). Classification of packed executables for accurate computer virus detection. *Pattern recognition letters*, *29*(14), 1941-1946.

[73] You, I., & Yim, K. (2010, November). Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications* (pp. 297-300). IEEE.

[74] Subrahmanian, V. S., Ovelgönne, M., Dumitras, T., & Prakash, B. A. (2015). Types of

malware and malware distribution strategies. In *The Global Cyber-Vulnerability Report* (pp. 33-46). Springer, Cham.

[75] Lapowsky, I. (2020). Malware last 10 years. *AV-TEST, shorturl. at/yzN01*.

[76] Weaver, N., Paxson, V., Staniford, S., & Cunningham, R. (2003, October). A taxonomy of computer worms. In *Proceedings of the 2003 ACM workshop on Rapid Malcode* (pp. 11-18).

[77] Kushner, D. (2013). The real story of Stuxnet. IEEE *Spectrum*, *50*(3), 48-53.

[78] Langner, R. (2011). Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, *9*(3), 49-51.

[79] Sung, A. H., Xu, J., Chavez, P., & Mukkamala, S. (2004, December). Static analyzer of vicious executables (save). In *20th Annual Computer Security Applications Conference* (pp. 326-334). IEEE.

[80] Orman, H. (2003). The Morris worm: A fifteen-year perspective. *IEEE Security & Privacy*, *1*(5), 35-43.

[81] Elisan, C. C., & Hypponen, M. (2013). *Malware, Rootkits & Botnets: A beginner's guide* (pp. 9-82). New York: McGraw-Hill.

[82] Everett, C. (2016). Ransomware: to pay or not to pay?. *Computer Fraud & Security*, *2016*(4), 8-12.

[83] Akbanov, M., Vassilakis, V. G., & Logothetis, M. D. (2019). WannaCry ransomware: Analysis of infection, persistence, recovery prevention and propagation mechanisms. *Journal of Telecommunications and Information Technology*.

[84] Akbanov, M., Vassilakis, V. G., & Logothetis, M. D. (2019). WannaCry ransomware: Analysis of infection, persistence, recovery prevention and propagation mechanisms. *Journal of Telecommunications and Information Technology*.

[85] Dai, A. M., & Le, Q. V. (2015). Semi-supervised sequence learning. *Advances in neural information processing systems*, *28*, 3079-3087.

[86] Almeida, F., & Xexéo, G. (2019). Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*.

[87] Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.

[88] Yan, M. (2019). Adaptive learning knowledge networks for few-shot learning. IEEE

Access, 7, 119041-119051.

[89] Sukhbaatar, S., Grave, E., Lample, G., Jegou, H., & Joulin, A. (2019). Augmenting self-attention with persistent memory. *arXiv preprint arXiv:1907.01470*.

[90] Kingma, D. P., & Ba, J. (2015, January). Adam: A Method for Stochastic Optimization. In *ICLR (Poster)*.

[91] Loshchilov, I., & Hutter, F. (2018, September). Decoupled Weight Decay Regularization. In *International Conference on Learning Representations*.

[92] Ma, J., & Yarats, D. (2021, May). On the Adequacy of Untuned Warmup for Adaptive Optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence* (Vol. 35, No. 10, pp. 8828-8836).

[93] LEBRET, R. P. (2016). *Word Embeddings for Natural Language Processing* (Doctoral dissertation, ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE).

[94] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, *15*(1), 1929-1958.

[95] Fan, A., Grave, E., & Joulin, A. (2019, September). Reducing Transformer Depth on Demand with Structured Dropout. In *International Conference on Learning Representations*.

[96] Tokenizer source code, last accessed on https://github.com/python/cpython/blob/3.10/Lib/tokenize.py

[97] Ba, J., & Frey, B. (2013). Adaptive dropout for training deep neural networks. *Advances in neural information processing systems*, *26*, 3084-3092.

[98] Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: A new framework for neural computation based on perturbations. *Neural computation*, *14*(11), 2531-2560.

[99] Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R., ... & Pal, C. J. (2017, January). Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations. In *ICLR (Poster)*.

[100] Zhang, Z., & Sabuncu, M. R. (2018, January). Generalized cross entropy loss for training deep neural networks with noisy labels. In *32nd Conference on Neural Information Processing Systems (NeurIPS)*.

[101] Ankile, L. L., Heggland, M. F., & Krange, K. (2020). Deep Convolutional Neural

Networks: A survey of the foundations, selected improvements, and some current applications. *arXiv preprint arXiv:2011.12960*.

[102] Alhaag, A. A., Savic, G., Milosavljevic, G., Segedinac, M. T., & Filipovic, M. (2018). Executable platform for managing customizable metadata of educational resources. *The Electronic Library*.

[103] Pietrek, M. (2002). An in-depth look into the Win32 portable executable file format, part 2. *MSDN Magazine, March*.

[104] Pefile, last accessed on https://github.com/erocarrera/pefile

[105] Sennrich, R., Haddow, B., & Birch, A. (2016, August). Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (pp. 1715-1725).

[106] Virustotal, last accessed on https://www.virustotal.com

[107] Das Malwerk, last accessed on https://dasmalwerk.eu/

[108] Panek, W., & Wentworth, T. (2010). *Mastering Microsoft Windows 7 Administration*. John Wiley & Sons.

[109] Schaefer, K., Cochran, J., Forsyth, S., Glendenning, D., & Perkins, B. (2012). *Professional Microsoft IIS 8*. John Wiley & Sons.

[110] Bott, E., & Stinson, C. (2019). *Windows 10 inside out*. Microsoft Press.

[111] Peng, P., Yang, L., Song, L., & Wang, G. (2019, October). Opening the blackbox of virustotal: Analyzing online phishing scan engines. In *Proceedings of the Internet Measurement Conference* (pp. 478-485).

[112] Cheng, B., Ming, J., Fu, J., Peng, G., Chen, T., Zhang, X., & Marion, J. Y. (2018, October). Towards paving the way for large-scale windows malware analysis: Generic binary unpacking with orders-of-magnitude performance boost. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 395-411).

[113] Kim, D., Kwon, B. J., Kozák, K., Gates, C., & Dumitraș, T. (2018). The Broken Shield: Measuring Revocation Effectiveness in the Windows Code-Signing {PKI}. In *27th {USENIX} Security Symposium ({USENIX} Security 18)* (pp. 851-868).

[114] Catakoglu, O., Balduzzi, M., & Balzarotti, D. (2016, April). Automatic extraction of indicators of compromise for web applications. In *Proceedings of the 25th international*

conference on world wide web (pp. 333-343).

[115] Chandra, R. V., & Varanasi, B. S. (2015). *Python requests essentials*. Packt Publishing Ltd.

[116] Nair, V. G. (2014). *Getting started with beautiful soup*. Packt Publishing Ltd.

[117] Fan, J., Wang, Z., Xie, Y., & Yang, Z. (2020, July). A theoretical analysis of deep Q-learning. In *Learning for Dynamics and Control* (pp. 486-489). PMLR.

[118] Han, M., Zhang, L., Wang, J., & Pan, W. (2020). Actor-critic reinforcement learning for control with stability guarantee. *IEEE Robotics and Automation Letters*, *5*(4), 6217-6224.

[119] Liang, G., Pang, J., & Dai, C. (2016). A behavior-based malware variant classification technique. *International Journal of Information and Education Technology*, *6*(4), 291.

[120] Mohaisen, A., & Alrawi, O. (2013, May). Unveiling zeus: automated classification of malware samples. In *Proceedings of the 22nd International Conference on World Wide Web* (pp. 829-832).

[121] Mohaisen, A., Alrawi, O., & Mohaisen, M. (2015). AMAL: high-fidelity, behavior-based automated malware analysis and classification. *computers & security*, *52*, 251-266.

[122] Galal, H. S., Mahdy, Y. B., & Atiea, M. A. (2016). Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, *12*(2), 59-67.

[123] Ki, Y., Kim, E., & Kim, H. K. (2015). A novel approach to detect malware based on API call sequence analysis. *International Journal of Distributed Sensor Networks*, *11*(6), 659101.

[124] Fan, C. I., Hsiao, H. W., Chou, C. H., & Tseng, Y. F. (2015, July). Malware detection systems based on API log data mining. In *2015 IEEE 39th annual computer software and applications conference* (Vol. 3, pp. 255-260). IEEE.

[125] Zeidanloo, H. R., Tabatabaei, F., Amoli, P. V., & Tajpour, A. (2010, July). All About Malwares (Malicious Codes). In *Security and Management* (pp. 342-348).

[126] Wang, A., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. (2018, November). GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP* (pp. 353-355).

[127] Dai, Z., Yang, Z., Yang, Y., Carbonell, J. G., Le, Q. V., & Salakhutdinov, R. (2019, January). Transformer-XL: Attentive Language Models beyond a Fixed-Length Context.

In *ACL (1)*.

[128] Bojan Kolosnjaji, Apostolis Zarras, George Webster and Claudia Eckert. Deep Learning for Classification of Malware System Call Sequences. In: Kang B., Bai Q. (eds) AI 2016: Advances in Artificial Intelligence. AI 2016. Lecture Notes in Computer Science, vol 9992. Springer Verlag.

[129] Kim, K., Aminanto, M. E., & Tanuwidjaja, H. C. (2018). *Network Intrusion Detection Using Deep Learning: A Feature Learning Approach*. Springer.

[130] Jiang, Z., & Zhang, S. (2020, May). Research on Task-oriented Dialogue Based on Modified Transformer. In Journal of Physics: Conference Series (Vol. 1544, No. 1, p. 012188). IOP Publishing.

[131] Krueger, D., Maharaj, T., Kramár, J., Pezeshki, M., Ballas, N., Ke, N. R. & Pal, C. J. (2017, January). Zoneout: Regularizing RNNs by Randomly Preserving Hidden Activations. In ICLR (Poster).

[132] Andrews, L. C. (1998). Special functions of mathematics for engineers (Vol. 49). Spie Press.

[133] Butcher, J. C. (2016). Numerical methods for ordinary differential equations. John Wiley & Sons.

[134] Rousseeuw, P. J., & Croux, C. (1993). Alternatives to the median absolute deviation. Journal of the American Statistical association, 88(424), 1273-1283.

[135] Deisenroth, M. P., Faisal, A. A., & Ong, C. S. (2020). Mathematics for machine learning. Cambridge University Press.

[136] Li, C. K., Rodman, L., & Šemrl, P. (2002). Linear transformations between matrix spaces that map one rank specific set into another. Linear algebra and its applications, 357(1-3), 197-208.

[137] Grinstead, C. M., & Snell, J. L. (1997). Introduction to probability. American Mathematical Soc.

[138] Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *arXiv preprint arXiv:1607.06450*.