Master Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Malware detection based on call graph similarities

Bc. Štěpán Dvořák

Supervisor: Ing. Danila Khikhlukha, Ph.D.
May 2020

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení:	**Dvořák**	Jméno: **Štěpán**	Osobní číslo: **435012**
Fakulta/ústav:	**Fakulta elektrotechnická**		
Zadávající katedra/ústav:	**Katedra počítačů**		
Studijní program:	**Otevřená informatika**		
Specializace:	**Kybernetická bezpečnost**		

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Detekce škodlivých souborů na základě podobnosti grafu volání funkcí

Název diplomové práce anglicky:

Malware detection based on call graph similarities

Pokyny pro vypracování:

1. Review current methods for detection of malicious PE/PE32 files (executable Windows files) and identify their problems.
2. Analyze information in disassembled binaries and propose their numerical representation.
3. Propose a method for classification of PE files.
4. Test proposed solution on a representative dataset and compare it with existing methods.

Seznam doporučené literatury:

[1] Le, Quoc, and Tomas Mikolov. &quot;Distributed representations of sentences and documents.&quot; International conference on machine learning. 2014.
[2] Yang, Zichao, et al. &quot;Hierarchical attention networks for document classification.&quot; Proceedings of the 2016 conference of the North American chapter of the association for computational linguistics: human language technologies. 2016.
[3] Gandotra, Ekta, Divya Bansal, and Sanjeev Sofat. &quot;Malware analysis and classification: A survey.&quot; Journal of Information Security 5.02 (2014): 56.
[4] Alon, Uri, et al. &quot;code2vec: Learning distributed representations of code.&quot; Proceedings of the ACM on Programming Languages 3.POPL (2019): 40.
[5] Xu, Ming, et al. &quot;A similarity metric method of obfuscated malware using function-call graph.&quot; Journal of Computer Virology and Hacking Techniques 9.1 (2013): 35-47.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Danila Khikhlukha, Ph.D., katedra počítačů FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2020** Termín odevzdání diplomové práce: **22.05.2020**

Platnost zadání diplomové práce: **30.09.2021**

Ing. Danila Khikhlukha, Ph.D. podpis vedoucí(ho) práce	podpis vedoucí(ho) ústavu/katedry	prof. Mgr. Petr Páta, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.	
Datum převzetí zadání	Podpis studenta

Acknowledgements

I want to thank my supervisor Ing. Danila Khikhlukha, Ph.D., for his valuable guidance and advice. Discussions with him always opened up new directions for the work.

I also appreciate valuable comments from doc. Ing. Tomáš Pevný, Ph.D. that helped me shape the thesis.

I am grateful to the Cognitive Intelligence team at Cisco Systems, Inc. for providing me with the opportunity and resources to work on this project.

My thanks also go to my family and my wife Tereza for all their support.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses. Prague, 10. May 2020

Abstract

Machine learning-powered malware detection systems became a necessity to fight the rising volume of malware. Malware authors create more sophisticated programs to overcome always improving antivirus engines. Windows OS remains the most targeted system, and the malicious payload commonly comes in the *Portable executable* (PE) file format. PE files can be analyzed with the *static analysis* methods, which are suitable for processing large amounts of data. Many engines disassemble binaries and study the code, which carries valuable insight into binary behavior. The assembly code is divided into functions that carry the functionality. The relations between functions form a Function Call Graph (FCG). FCG has been studied in the literature, and the graph structure was employed to find similarities between files. Recently, Graph Neural Networks (GNNs) have been adapted to work upon FCGs and are claimed to be performing well. In this work, we study and compare different GNN models and their architectures. After selecting the best GNN model, we compare it with a non-structural model to verify if an FCG structure improves classification models. We perform our empirical study on a large dataset of more than 5 million PE files.

Keywords: static analysis, malware classification, function call graph, graph neural networks

Supervisor: Ing. Danila Khikhlukha, Ph.D.

Abstrakt

S rostoucím množstvím škodlivých souborů se stalo využití strojového učení pro jejich detekci nezbytností. Autoři škodlivých souborů vytváří důmyslnější programy, aby překonali stále se zlepšující antivirovou ochranu. Windows OS zůstává nejčastějším cílem útoků. Viry se často šíří ve formátu Portable Executable (PE). PE soubory mohou být zkoumány pomocí metod statické analýzy, které se hodí pro zpracovávání velkého množství dat. Mnoho antivirových systémů disassembluje soubory a zkoumá jejich kód, který nabízí vhled do funkcionality souboru. Assembly kód je členěn do funkcí. Vztahy mezi funkcemi zachycuje graf volání funkcí (GVF). Tento graf byl zkoumán v literatuře a jeho struktura byla využita k hledání podobností mezi soubory. V poslední době začaly být úspěšně využívány grafové neuronové sítě (GNN) ke zpracování těchto grafů. V naší práci zkoumáme různé druhy a architektury GNN a vzájemně je porovnáváme. Po tom, co vybereme nejlepší GNN model, ho srovnáme s modelem, který nevyužívá grafovou strukturu GVF, abychom zjistili zda tato struktura zlepšuje klasifikační modely. Naši studii provádíme na velkém datasetu o více než 5 milionech PE souborů.

Klíčová slova: statická analýza, klasifikace škodlivých souborů, graf volání funkcí, grafové neuronové sítě

Překlad názvu: Detekce škodlivých souborů na základě podobnosti grafu volání funkcí

Contents

Figures

Tables

Chapter 1

Introduction

According to AV-test [1], the total number of malware in the wild reached 903.14 million samples in May 2019 (recent numbers are shown in Fig. 1.1). The number is still rising, with 376,639 new samples detected on average per day. As in previous years, Windows OS is the most targeted system, with 51.08% of the total malware samples in 2018. Portable executable (PE) is the third most common file type used in malicious emails [4]. Malicious programs keep getting more sophisticated to overcome continually improving detection engines defending users. The attackers target sensitive data, steal bank account credentials, blackmail victims after encrypting their data, use the computation power of the victims for coin mining [1]. While financial gain is the most common (71%) motivation of attacks, espionage is the motive in 25% cases, as stated by Verizon [5]. We see that cyberattacks are a threat to our computers as well as for whole countries.

It is necessary to develop systems that would protect computers from the ever-rising threat of a cyber attack. These systems should be able to work on a large scale to process the large volume of potentially malicious files. Machine learning methods have been studied extensively and successfully adapted to this task. *Static analysis* methods aim to detect malicious files without actually executing them. This approach can process vast amounts of data [6]. *Dynamic analysis* executes files in a controlled environment to capture their behavior. The execution traces are then analyzed to discover malicious actions. However, the execution is too expensive and time-consuming to be performed on large volumes of data. It is also necessary to create execution scenarios that would expose the malicious activity [6]. Therefore, we focus on static analysis systems in our work.

Static malware detection systems usually use features extracted from PE header of the file, strings found in bytes of the executable, byte sequences, etc. [6]. There are also approaches discovering the functionality of an executable by disassembling it [7] [8] [9]. The disassembled code can be further analyzed and scanned for malicious patterns. Researchers also studied *Function Call Graphs* (FCGs) [10][11][12][13], which capture the structure of a program and relations between its functions. Many of those approaches used manually

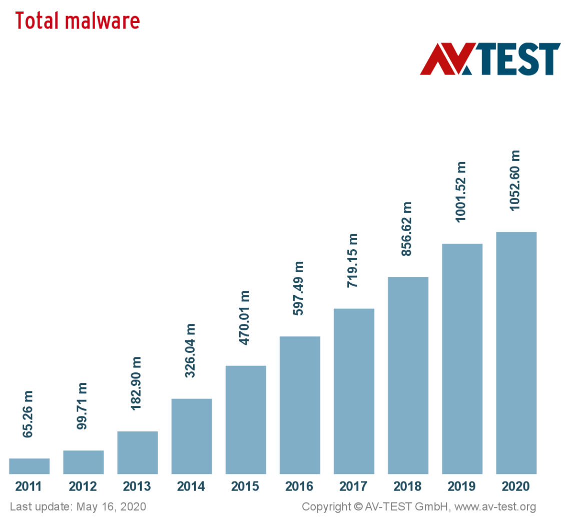


Figure 1.1: Total number of malware as observed by AV-test. [1]

designed heuristics to compute the similarity between graphs. These also could not scale to large volumes of data and large graphs [14], which was the reason for the adoption of *Graph Neural Networks* (GNNs) [15] [14]. GNNs provide state-of-the-art performance in many tasks [16] [17]. They exploit the structure of a graph and propagate information from each node through the graph.

Recent works claimed that GNNs could exploit a structure of an FCG and use it to produce accurate verdicts for files [14] [15]. In the GNN, each function gets a context of functions calling it so that it can create better representations of functions and consequently, the whole files. We want to use GNNs to find similarities between FCGs because we expect the graphs to be similar within e.g., malware families, or minor updates of the same applications. We try to replicate successes of GNNs and verify that the graph structure is indeed beneficial for the classification task. We test the models on a large set of real-world data with more than 5 million samples. We discuss and compare different variants of GNNs and show their ability to model PE files. Finally, we empirically prove that unexpectedly the structure-agnostic model produces better results than GNN, even at a lower computational cost. We believe that our work will encourage more researchers to investigate this phenomenon.

This thesis is organized as follows: In Chapter 2, we describe the structure of the PE file format. We explain how we can obtain an assembly code of a binary and which obstacles we can face during the task.

In Chapter 3, we review articles published about malware classification. We focus on machine learning methods, modeling of code, and graph neural networks.

In Chapter 4, we define all terms needed for graph tasks and further explain details about various GNN models.

In Chapter 5, we state our requirements on a dataset, review public datasets, and finally introduce our dataset together with its analysis.

In Chapter 6, we introduce models used in this work. We give details about the features and architectures that we use.

In Chapter 7, we study the proposed models. We compare various architectures of GNN models and compare their classification performance as well as their training time. After designing the GNN model, we compare it to the non-structural model.

We conclude this thesis in Chapter 8 and add suggestions for future work.

Chapter 2

Portable executable file

Almost all files with executable code are distributed in the form of a Portable Executable (PE) file format on Windows OS. It is also commonly used for spreading malware. PE files provide us several opportunities to detect their usage for malicious intent. On the other hand, malware authors are presented with the means to thwart the analysis of their programs. We describe the PE format in detail to describe the challenge we are facing. Firstly, we show how the PE format is defined. Secondly, we explain the machine code that forms the executables and its representation in human-readable form. Finally, we describe techniques that can be incorporated by malware authors to evade automatic analysis.

2.1 PE format

Several file types come in a PE format. These are:

- exe files - executables
- dll files - dynamic-link libraries which are supposed to be used by other programs
- object code - code generated by a compiler

The structure of a PE file is depicted in Fig. 2.1. We can see that it starts with headers. *PE header* provides necessary information for a Windows OS loader. The loader starts when a file is executed. It allocates memory for the data of the executable, resolves imported libraries and passes execution to the program.

PE header is a valuable source of information for an analyst. Interesting fields in the header are:

- Time Date Stamp - the time when a binary was compiled
- Subsystem - indicator whether a binary is a console or GUI program

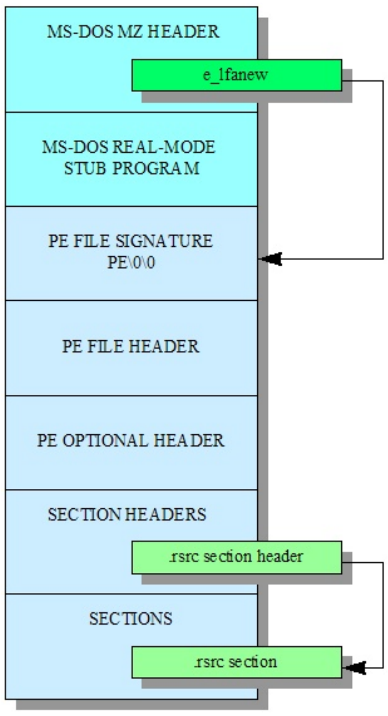


Figure 2.1: Structure of PE file [2].

- Section headers - information about each section including name and size
- Imports - imported functions from other libraries
- Exports - exposed functions to be called by other files

There are more interesting fields in the header that can be successfully used to detect malicious files, as shown in our previous work [18].

Analyzing a PE header is a good start when analyzing a binary, which helps to estimate its functionality. However, we usually need to dig deeper into a binary to get the complete picture. This is where analyzing its code comes into play.

2.2 Assembly language

The source code of a PE file might be a valuable source of information as it helps us understand its functionality. In the case of dynamic analysis, we need to craft inputs and scenarios to trigger events. Alternatively, when having the actual code, we can go through it and discover the complete functionality of the executable. We introduce a human-readable language that can be obtained from the raw data of a PE file. We start by reviewing basic concepts of programming languages with emphasis on assembly language. For more detailed explanation see [3].

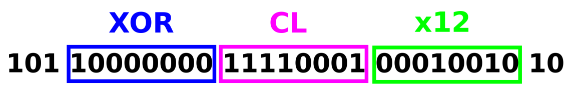


Figure 2.2: Machine code and its translation into assembly instruction "XOR CL 12".

2.2.1 Abstraction levels

A computer system is comprised of several levels of abstraction. These levels make it easier to develop programs by abstracting their code from the underlying hardware.

In our problem, these levels are important:

- *Machine code* - Consists of binary data that represent operations in the processor. It is further translated for the processor into *microcode* better known as firmware. Machine code is created during *compilation* from high-level languages.
- *Assembly language* - Human-readable language representing processor instructions.
- High-level language - Provides abstraction from the hardware level. Includes C, C++, etc.

2.2.2 Assembly elements

Although there are many dialects of assembly language (x86, x64, MIPS, ARM, etc.), we focus on x86. It is the most popular architecture, and Windows OS is designed to run on x86. Therefore, malware authors target x86 as well. We provide details of its elements.

Instructions

Instructions are the building block of the language, and they correspond to machine code instructions - *opcodes*. An instruction has a *mnemonic* and zero or more *operands*. A mnemonic is an operation to be performed, and operands are its arguments. An example of instruction is in Fig. 2.2. It performs `xor` operation between a number and a value in the register `cl`.

Operands can identify three types of data used by the instruction:

1. Immediate - numeric constant value, e.g. `0x11`
2. Register - e.g. `eax`
3. Memory address - reference to a location in memory, e.g. `[ESI+EAX]`

Register is fast storage available to CPU. A program is faster when it does not have to load data from RAM or drive but instead saves data in registers and loads them fast from there. There are four kinds of registers [3]:

1. General registers - Used by the program during its execution. They store data or addresses.
2. Segment registers - Used to keep track of currently used segments of data.
3. Status flags - Contain the current status of the CPU.
4. Instruction pointers - Points to the next instruction to be executed.

We give examples of some frequently used instructions:

- `mov` *destination source* - copies data from destination to source
- `lea` - load effective address
- `sub` *destination value* - subtract value from value at destination
- `je` *loc* - jump to location, occurs after `cmp` instruction because it depends on its result and flags set by it

There are hundreds of instructions in x86 architecture we refer an interested reader to [19].

Functions

A *function* is a sequence of instructions that performs a specific task. Stack and registers are prepared before the function call and restored after the function finishes. These procedures are subject to conventions.

1. Push arguments on the stack
2. Return address is stored on the stack
3. The function is called using `call` instruction
4. Space is allocated for local variables
5. Function runs its code
6. Stack is restored, local variables are freed
7. The function returns by calling `ret` instruction.

We omitted details about the allocation/deallocation of the memory. We can see that functions form specific constructs within the assembly code. Undoubtedly, it is beneficial to detect them in order to analyze the application in a more structured fashion, which eases the analysis significantly.

Function Call Graph

Function Call Graph (FCG) represents calling relations between functions. A node in the graph represents a function. An edge (e, j) indicates that function e calls function j. FCG is a valuable source of information because it gives us further details of how functions interact between themselves. We can examine in which contexts a function is called and whether it affects its meaning.

2.2.3 Disassembling

We usually have only a binary form of the executable, i.e., machine code. Deriving assembly code out of it is called *disassembling*. There are several challenges for a disassembler:

- Any inaccuracy in reading and interpreting bytes can result in a completely different outcome. This is a problem when locating an entry point or in case of variable length instructions.
- Malware writers are proactively trying to evade disassembling of their programs.
- Disassembler should run without the actual execution of the file. If we run the code, we could unintentionally break the computer when running the disassembly.

There are several disassemblers published that can be comfortably used. They all use some variations of two disassembly algorithms. We describe them and point to their weaknesses. Then, we describe several anti-disassembly techniques.

Linear disassembly

The *linear disassembly* or *linear sweep* iterates over bytes disassembling instructions one by one linearly. This approach will generally disassemble too much code. The main drawback is that it cannot differentiate between code and data. If there is data mixed with a code, it will try to disassemble the data, resulting in nonsense instructions.

Flow-oriented disassembly

The *flow-oriented disassembly* follows execution paths in the code. It does so to avoid erroneous disassembling of data as in the case of linear disassembly. The complicated nature of a program causes these errors. Linear disassembly assumes that instructions are stacked one after another, but it is not the case. Common binaries contain jump instructions, pointers, exceptions, and conditional branching. All these constructs break the linear nature of the program. So when the algorithm encounters jump or branch instruction during a linear sweep, it follows it and disassembles its destination. After that, it can continue in the linear sweep.

Binary ninja disassembler

We use Binary Ninja [20] disassembler in this work. It is a reverse engineering tool developed by Vector 35. It uses a combination of previously mentioned algorithms to perform disassembly. Unfortunately, vendors do not disclose details about their disassemblers, and Binary Ninja gives only a few details. In their blog post [21] they give some details about techniques used to detect functions:

- Recursive Descent - flow-oriented disassembly starting from defined entry points
- Call Target Analysis - they aggregate destination of a call instructions which are then passed to the recursive descent algorithm for analysis
- Control Flow Graph Analysis - analysis of Control Flow Graph and intraprocedural control-flow constructs bring new entrypoints that are then passed to the recursive descent.
- Tail Call Analysis - helps discover tail call functions used as an optimization in the compiler

They do not disclose other details publicly. However, we checked with analysts that their disassembly listings are accurate.

2.2.4 Anti-disassembly techniques

Malware authors do not want their programs to be analyzed easily. In fact, they want to make (at least automatic) analysis as difficult as possible. We give some examples of anti-disassembly techniques taken from [3]:

- Jump Instructions with the Same Target - this technique is illustrated in Fig. 2.3. We can see two instructions `jz` and `jnz` with the same target. Together they form an unconditional jump to `loc_4011C4+1`. The key part is `+1`. It points at the second byte of the call instruction. A basic disassembler will continue disassembling bytes after the two jump instructions resulting in `call` instruction. However, the real instruction which is executed starts with the byte `0xE8` which corresponds to `pop eax` instruction.
- A Jump Instruction with a Constant Condition - this technique is illustrated in Fig. 2.4. We can see a conditional jump `jz`. However, there is a `xor eax, eax`, which will always result in zero. Therefore, the condition

```
74 03                   jz      short near ptr loc_4011C4+1
75 01                   jnz     short near ptr loc_4011C4+1
                        loc_4011C4:                     ; CODE XREF: sub_4011C0
                                                        ; ❷sub_4011C0+2j
E8 58 C3 90 90        ❶call     near ptr 90D0D521h
```

Figure 2.3: An example of antidisassembly technique: Jump Instructions with the Same Target. [3]

```
33 C0                   xor     eax, eax
74 01                   jz      short near ptr loc_4011C4+1
        loc_4011C4:                             ; CODE XREF: 004011C2j
                                                ; DATA XREF: .rdata:004020ACo
E9 58 C3 68 94          jmp     near ptr 94A8D521h
```

Figure 2.4: An example of antidisassembly technique: A Jump Instruction with a Constant Condition. [3]

will always be satisfied, and the jump will be taken. As in the previous case, the destination of the jump is `loc_4011C4+1` that is byte `0xE8`. Usually, disassemblers start with the false branch of a jump instruction, so they will disassemble a `jmp` instruction first. In fact, they should start disassembling at the next byte, which is `0x58`. The correct instruction would be `pop eax`.

- Impossible Disassembly - this technique is shown in Fig. 2.5. The example starts with a two-byte instruction `jmp -1`, which points to the start of another instruction `inc eax`. We can see that the same bytes can be used in multiple contexts, and each time they can have a completely different meaning. Unfortunately, these situations cannot be handled in disassemblers. We cannot interpret a single byte as multiple instructions. See [3] for a more elaborate example. These situations require the work of an analyst or carefully designed macros that would clean the code.

We have presented examples of techniques to overcome automatic disassembly routines. Samples incorporating anti-disassembly techniques demand the work of an analyst, which is generally more expensive than an automated analysis. These examples exploited assumptions that disassemblers make during their work and the fact that instructions have multiple bytes.

We do not have any means to overcome the issues described in this section. We can only trust the disassemblers and choose the ones that are maintained and up-to-date with the current state of malware. We point to these problems to stress the limitations of our approach. Although we cannot fight them directly, we can design our methods to detect them. We also should be aware of these cases when analyzing results and misclassified samples that we are going to get.

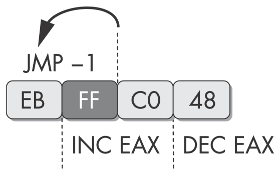


Figure 2.5: An example of antidisassembly technique: Impossible Disassembly. [3]

2.2.5 Binary packing

Another obstacle in the way of static analysis is a *binary packing*. Packers are used both in legitimate and malicious software. The reasons are:

- Size shrinkage
- Analysis prevention

Packers transform a program into a new one storing the original application as data. If we look at a packed binary, we see only an *unpacking stub*, and any analysis can be worthless. We show how binaries are packed and the possibility of unpacking. Although we do not perform unpacking in our work, we believe it is necessary to mention it as a limitation not only of our method but all static analysis methods.

Unpacking stub

An executable is wrapped by a different program called unpacking stub during a process called packing. When the unpacking stub is executed:

1. Unpacks the original binary into memory
2. Resolve all the imports
3. Pass the execution to the original binary

The original binary is stored as data and can be additionally compressed and even encrypted. Therefore, first, the stub needs to load the original binary into memory and perform decryption or decompression if necessary. Malware authors sometimes hide imported functions from the analyst, so the Import Address Table (IAT) in the PE header contains only a few or none imports. The stub needs to resolve all the imports, or it needs to reconstruct the IAT. Finally, the execution is forwarded to the original binary.

Packed binaries challenge

We explained how a packed binary looks. It is obvious that analyzing a packed binary without unpacking can be problematic.

There are several approaches to the challenge:

1. Treat a packed binary as any other - When we disassemble a packed binary, we will get the unpacking stub code. Analyzing the stub can yield a reasonable result, e.g., if the packer is characteristic for a malware family. Then, detecting the unique packer can provide the ability to detect the malware family. In the case of commonly used packers, analyzing the stub will not provide enough information to detect maliciousness of the file.

2. Detect packed files and filter them out - Packed binaries can be left for a dynamic analysis or a manual analysis by an analyst. However, detecting binary packing is not a trivial task, as well. There exist signatures that can help to detect common packers [22]. Other methods can be based on a combination of symptoms, e.g., higher entropy of sections, only a few imports, a small amount of code recognized, etc.

3. Detect a packer and perform unpacking - This should be the preferred approach. However, unpacking is difficult and may require manual analysis, which is unbearable on a large scale.

We choose the first approach in this work. Packer detection and unpacking are problems that we leave for future work.

Unpacking

Unpacking transforms a packed binary into the original one. It is necessary when we want to analyze the real behavior of the file. If we analyzed the packed binary, we would more likely be analyzing the unpacking stub. Unpacking can be automated in some cases, but generally, it is a difficult and expensive process.

There are 3 types of unpacking [3]:

1. Automated static unpacking - Fastest method, which automatically decompresses the executable. It does not execute the binary. Possible, for example, for UPX packer.

2. Automated dynamic unpacking - The unpacker runs the executable and waits for the unpacking stub to finish. However, it is difficult to recognize the end of the stub and the beginning of the original binary. It is also necessary to run the unpacking in a safe and controlled environment.

3. Manual unpacking - This approach needs to be chosen when the previous ones failed. In this case, an analyst needs to detect an algorithm that was used for the packing. After that, he/she can design a program that can revert the packing. Another approach is to run the unpacking stub and wait for it to finish. Then, it is possible to dump the executable from memory.

We described ways of unpacking executables. Since we are designing a system that can work on scale, we would need an automated unpacker. However, we did not find a publicly available unpacker that would suit our needs. Therefore, we leave unpacking for future work.

Chapter 3

Related work

It is necessary to develop malware detection systems that can handle large volumes of data since the total number of malware is continuously rising [1]. Researchers addressed this issue in numerous works [6]. We focus on works using performing static analysis, i.e., without actual execution of the file. We start by reviewing various malware detection methods, and we end up at methods that use neural networks to model assembly code and exploit Function Call Graph (FCG) structure.

3.1 Malware detection

Detecting malware using static analysis methods has been addressed in many works during the last decade. We review methods on different levels of a binary. We start by models that operate on raw binary code, then move to assembly code that can be obtained via disassembling. Finally, we summarise approaches that make use of FCGs.

3.1.1 Raw bytes approach

The problem of learning malicious patterns in raw bytes of files has been addressed in numerous papers. Raff et al. [23] proposed training Convolutional Neural Network on the full binary content of a file. Their largest model was trained on a dataset of 2 million samples and had accuracy 94.0 on a test set of 43,967 malicious and 21,854 benign files. The training took one month on a system with 8 GPUs.

Jain et al. [24] proposed selecting the most relevant byte n-grams sequences and training various classifiers upon them. The most relevant byte n-grams are chosen based on classwise document frequency. They tested their approach on a dataset of 1,018 malicious and 1,120 benign samples. They obtained an accuracy of 99.0 and found out that classifiers perform the best on 3-grams. Random forest was the best performing model in their work outperforming Naive Bayes, Instance-Based learner, Decision Tree, AdaBoost.

3.1.2 Opcodes approaches

Many articles go beyond the simple ingestion of raw bytes and operate on operational codes (opcodes) sequences. Santos et al. [7] proposed a framework that studies the possibility of using opcode sequences for malware detection. They claim that bytes based models can be fooled by code obfuscation. Therefore, they extracted 1-grams and 2-grams of opcodes from each file and selected 1000 the most relevant ones based on Information Gain. They trained several classifiers on these features, including k-NN, Decision Tree, SVM, Bayesian networks. They tested their method on a dataset of 17,000 malicious files, including 585 malware families, and 1,000 benign files. They tested different lengths of n-grams, for n = 1 there were 348 different opcode sequences, 51,949 for n = 2, 1,360,744 for n = 3, and 10,309,792 for n = 4. However, they completed the feature selection step only for n=1 and n=2. Better results were obtained for longer n-grams, and almost every classifier had accuracy above 90%.

Nagano et al. [25] tried natural language processing method to model disassembly listings, imports, and hexdump. They trained a paragraph vector of dimension 100 for each feature category. K-nearest neighbor (KNN) and SVM classifiers were trained on these features. The approach was tested on a dataset of 3600 binaries using 10-fold cross-validation. They measured the performance of each feature separately as well as all combined. KNN provided superior results when used on each set of features separately, having the highest accuracy at 0.995278 on a paragraph vector of assembly code. SVM had slightly better results when all features were concatenated. The accuracy was 0.9944, precision 0.9940, recall 0.9950 and F-measure 0.9945.

Lu [9] proposes using LSTM to model opcodes. They disassemble executables using IDA Pro and extract their opcodes. However, they extracted assembly code only from a `.text` section. Opcodes embeddings are obtained by training a paragraph vector. These vectors are fed into two-stage LSTM, and a model is trained. The first LSTM layer consumes opcodes of each particular function, whereas the second layer works upon the output of the first LSTM layer. Outputs of the second-layer LSTM are averaged and used as an input to softmax. Their approach was tested on a dataset of 969 malware and 123 benign samples. Their best performing model on binary classification task has accuracy 97.87% and 94.51% on multi-class classification.

Kolosnjaji et al. [8] utilized Convolutional Neural Networks (CNN) for malware detection. They combined features extracted from PE header, imported functions, and opcodes. Imported functions and opcodes are one-hot encoded, whereas PE header features are real values. PE header features, together with import features, are fed into a feedforward neural network. The CNN processes opcodes with filter height 3, and the result is run through the max-pooling subsampling layer. They tested their approach on a dataset of 22,694 malicious and 63 legitimate samples. The model learned to predict 14 labels where labels correspond to clusters of malicious binaries, and one label stands for legitimate samples. They obtained precision and recall 0.93 with

F1-score 0.92.

3.1.3 Function call graph approaches

Various works use the similarity of Function Call Graphs (FCG) for malware detection. FCGs are usually extracted using static analysis after disassembling a binary. In many tasks, it is useful to compare FCGs for which a similarity metric is needed. Many works use Graph Edit Distance (GED) as a metric for comparing graphs. GED measures a minimum number of edit operations required to transform one graph into another. However, GED is an NP-complete task, so approximation algorithms are needed.

Kinable et al. [10] presented an algorithm approximating GED and its usage for malware classification. They start the computation of GED by matching external functions. The remaining functions (nodes) are then matched using simulated annealing in a way that GED is minimized. The resulting GED is a normalized number of inserted/deleted vertices, a number of unpreserved edges, and a number of mismatched external functions. Later, they use their similarity metric to perform k-medoids and DBSCAN clustering and test it on a dataset of 194 malware samples from 24 families. They concluded that they successfully identified malware families.

Hu et al. [11] used similar approach as Kinable [10]. They use an approximation of GED distance to run queries for similar samples. Several tricks for speeding up GED computation were introduced. Firstly, they match library functions with the same names. Secondly, they match functions based on their mnemonic sequences. Thirdly, they match functions by computing edit distance between mnemonic sequences where functions above a certain threshold are matched. Finally, they match the remaining functions by running the Hungarian algorithm [26]. They also presented an efficient way of storing data to make the queries faster. Their experiments are more focused on the effectiveness of their approach but also provide results regarding their classification performance. They obtained a success rate of 80.10%, representing the ratio of queries that correctly found a sample from the same family among 5 returned samples.

Hassen et al. [12] uses clustered functions to compare FCGs of files. Firstly, they construct an FCG of a file and label each vertex. Vertices representing external functions are labeled by their name. However, local functions do not have any reasonable name because they were lost during compilation. They argue that edit distance between opcodes of two functions is a suitable metric, but its computational complexity is too high. Therefore, they compute a hash of each local function using Local Sensitivity Hashing. The resulting hash is used as a label for a function. We can think of the hashing as of the clustering of functions. Then, they create a vector for each graph by counting labels of vertices and corresponding edge types (determined by hashes of the two corresponding vertices) within the graph. They create a vectorized representation of a graph that reflects the FCG structure. Finally, they train a random forest classifier on their dataset of 10,260 samples from

Microsoft Malware Classification Challenge. They obtained an accuracy of 0.979 outperforming work of Kinable et al. [10] having 0.840.

Xu et al. [13] proposed heuristics for matching FCGs. The similarity score is computed based on a normalized number of common edges. Finding common edges happen in several steps:

1. Matching external functions based on their names
2. Matching local functions by the same called external functions - Functions are matched if they call more than two same external functions.
3. Matching local functions based on the opcodes - They divide opcodes to 15 categories based on functionality (data, string, loop, etc.). They extract color for each function (15-bit number each bit corresponding to one opcode category) and a vector with the number of instructions in each category. Then for each function with the same color, they compute the cosine similarity between their vectors. They also compute the similarity of lengths and degrees. If all those similarities are above specified thresholds, they match the functions.

Finally, the number of matched edges is computed, and its ratio to all edges is used as a similarity matrix. They tested their approach on several tasks: measuring similarity between malware variants, classification of different malware families, binary classification of malicious and legitimate binaries. They did not provide a single metric that could report their results but rather confusion matrices. They concluded that their approach could recognize malware variants and different malware families.

3.2 Code modelling

Yan et al. [15] proposes using Deep Graph Convolutional Neural Network (DGCNN) on the Control Flow Graph (CFG) for malware classification. CFG has basic blocks as vertices that are connected with an edge either by jump instruction or simply by consecutiveness in a code. Each node is described by 11 features, including a count of numeric constants, total instructions, degree, etc. Features are fed to the neural network and propagated through convolutional layers. Outputs of all convolutional layers are concatenated at the end, and AdaptiveMaxPooling Layer is applied. AdaptiveMaxPooling transforms the input into fixed-sized output by adapting stride and filter size to produce an output of a given size. Series of convolutional layers is applied on the output and fed into a fully connected layer with perceptron at the end. They also tested a different approach for pooling the data called SortPooling together with different methods to convert matrices into single vectors for each graph. However, those approaches had an inferior result. They tested their approach on Microsoft Malware Classification Challenge [27] (10,868 malware samples) and the YANCFG dataset [28] (16,351 malware samples) and obtained accuracy 99.25. They present only per family results on the YANCFG dataset.

Alon et al. [29] presented an approach of learning embeddings of a code. Their work focuses on learning vectors representing the semantic properties of code snippets. Abstract Syntax Tree (AST) is built for each code snippet, and it represents various possible paths in the code. A bag of path-contexts is extracted from an AST where a single path context is a pair of two terminals and a path between them. Each terminal and path has its embedding, and these three embeddings are concatenated together into a single context vector. All context vectors of a snippet are fed into a fully connected layer, which combines those three different embeddings. Then, an attention mechanism is applied on top, and a code vector is obtained. Attention weights are nonnegative and sum up to 1. They tested the approach on the task of predicting method names of Java methods. Their work significantly improved prediction scores over previous works.

Phan et al. [14] proposed a convolutional network working on labeled directed graphs called *DGCNN*. Their network operates on an FCG, and each node (function) has a representation based on its instructions. Two convolutional layers process the graph. Representations of particular nodes are aggregated using max-pooling. The resulting vector representing the graph is fed to two fully connected layers, and softmax to produce a label. They tested their approach on two tasks. The first one is software defect detection, and the second one is malware analysis. In case of malware analysis, their dataset has 2,937 samples. They obtained an accuracy of 0.9731 and average AUC 0.9722.

Chapter 4

Graph neural networks

Many algorithms have been proposed for various tasks on graphs. We focus on a class of algorithms called *Graph Neural Network* (GNN). It takes advantage of a structure of the graph while having the power of neural networks. GNNs were applied in various areas of research [16]:

- Structural tasks - Data has a graph structure naturally, e.g., social networks, recommender systems.
- Non-structural tasks - The graph relations are not explicitly in the data, e.g., images, text.
- Other tasks - Tasks such as building generative models.

In many cases, they brought a performance boost and proved to be efficient even on large graphs [17].

Depending on the task we are solving GNN models can be further distinguished by the level they operate on [17]:

- Node-level - node regression, node classification task
- Edge-level - edge classification, link prediction
- Graph-level - graph classification

In this chapter, we start by defining the terms needed to understand GNN. We further focus on the graph classification task and introduce Convolutional Graph Neural Networks. We describe different kinds of convolutional layers that we use in this work. We further explain what other layers are used in the GNN and how we can train a model. We study GNNs in detail so we will be able to build a reasonable model for our task.

4.1 Graph

Firstly, let us define a graph:

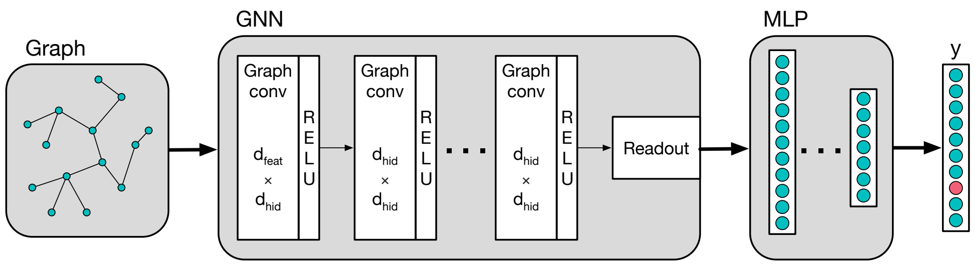


Figure 4.1: GNN model structure.

Definition 4.1. A Graph is a pair $G = (V, E)$ where V is a set of vertices (nodes) and E a set of edges.

We denote the total number of nodes in a graph as $|V| = n$.

Definition 4.2. A directed edge $e \in E$ is an ordered pair (u, v), where $u, v \in V$. An undirected edge is similarly an unordered pair of vertices.

Definition 4.3. A neighborhood of a vertex v is defined as $N(v) = \{u \in V|(u, v) \in E\}$.

So far, we have defined a general graph. We enrich the graph with node attributes (features):

Definition 4.4. A matrix $\mathbf{X} \in \mathbb{R}^{(n \times d)}$ is an attribute matrix. $x_v \in \mathbb{R}^d$ is a feature vector of a node v, and d is the number of features.

Graphs are used to represent many real-world problems because they capture relations between entities (nodes) in the graph.

4.2 Graph classification

We are focusing on graph-classification in this work, but the concepts described here can be transferred to other tasks as well. A general net architecture for graph classification task is in Fig. 4.1. The graph, as defined before, does not need any other preprocessing and enters as an input. We can see that we build the network by repeating a graph convolution, followed by an activation function.

Graphs in a dataset do not usually have the same number of nodes, so it is not possible to directly classify the whole graph. Each node has a vector representing its state, and we need to aggregate these vectors to get a single one representing the whole graph. Therefore, we have to add *readout* layer, which builds the graph representation. This layer can be followed by a multi-layer perceptron and a softmax layer to produce a graph-level label.

4.3 Convolutional graph neural networks

There are several types of neural networks operating on graphs, i.e., Graph Neural Networks (GNN). We have seen tremendous improvements in many fields due to Convolutional Neural Networks (CNN) and recently ConvGNN generalized *convolution* from grid data to graphs [17]. ConvGNNs gained attention due to their ability to capture relations in data and use them for a performance boost.

Standard CNN performs convolution upon two-dimensional matrices (e.g., images) where each element in the matrix is a pixel. The pixels are naturally order based on their appearance in the image. We can not reorder pixels in the image without losing the original image. There is a rigorous way of stacking the elements into the matrix representing a sample. However, there is not a natural order in a graph. We can arbitrarily rearrange the vertices in space, but the graph would remain the same. In order to linearize a graph into a matrix, we would have to explore several (if not all) possible orderings. GNNs tackle the problem by performing convolution on a node level while ignoring the ordering. Generally, GNN updates an embedding (hidden state) of a node v by aggregating embeddings of its neighbors $N(v)$. In that way, the information in graphs is propagated.

There are two types of ConvGNNs [17]:

- *Spectral-based ConvGNNs* perform convolution using filters known in signal processing. These filters require computation of eigenvalue decomposition of the graph Laplacian. The filter $\mathbf{g}_\Theta = \text{diag}(\Theta)$ can be written as:
$$\mathbf{g}_{\boldsymbol{\Theta}} * \mathbf{x} = \mathbf{U}\mathbf{g}_\Theta\mathbf{U}^T\mathbf{x}$$
where $\mathbf{U}$ is a matrix of eigenvectors of the graph Laplacian:
$$\mathbf{L} = \mathbf{I}_N - \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}} = \mathbf{U}\boldsymbol{\Lambda}\mathbf{U}^\mathbf{T}$$
where $\mathbf{D}$ is a diagonal matrix of node degrees $\mathbf{D} = \sum_j A_{i,j}$. $\mathbf{A}$ is an adjacency matrix and $\boldsymbol{\Lambda}$ is a diagonal matrix of eigenvalues.
Computing eigenvalue decomposition is many times not feasible because its complexity is $O(n^3)$. Therefore, an approximation of the filter is used instead. Another problem of spectral models is its dependence on the Laplacian eigenbasis. Any changes in the graph result in a change of eigenbasis, which causes the method to be dependent on a graph structure. It means that a model trained on a particular graph structure can not be directly used on a graph with a different structure [16]. This problem is again solved by approximation to make the method independent of the actual values of eigenvectors.
- *Spatial-based ConvGNNs* perform convolution directly on graphs using spatial relations of nodes. The representation of a node is obtained via aggregation of neighbors' representations where information is passed over edges.

4.3.1 Graph Convolution Layer

There are several variants of convolutional layers [17][16]. We give a description of particular variants that can be part of the network. We do not provide an exhaustive list but rather focus on those relevant to this work. For more complete overview of GNN variants see [17][16].

ChebConv

ChebConv [30] is a spectral-based ConvGNN model. It approximates the filter $\mathbf{g}_{\mathbf{\Theta}}$ by Chebysev polynomials up to K^{th} order. Chebysev polynomials are defined recursively as:

$$\mathbf{T}_k(\mathbf{x}) = 2\mathbf{x}\mathbf{T}_{k-1}(\mathbf{x}) - \mathbf{T}_{k-2}(\mathbf{x}),$$

$$\mathbf{T}_0(\mathbf{x}) = \mathbf{1}$$

$$\mathbf{T}_1(\mathbf{x}) = \mathbf{x}$$

We use them to approximate the filter:

$$\mathbf{g}_{\mathbf{\Theta}} * \mathbf{x} \approx \sum_{k}^{K} \Theta_k \mathbf{T}_k(\hat{\mathbf{L}})\mathbf{x}$$

where

$$\hat{\mathbf{L}} = \frac{2}{\lambda_{\max}}\mathbf{L} - \mathbf{I}_N$$

and $\lambda_{\max}$ is the largest eigen value of $\mathbf{L}$. The filters in ChebConv are localized in space so they can extract features independently of graph size and without the need of computing eigenvectors.

To transfer it into our task. The new representation of nodes $\mathbf{X}'$ is obtained as:

$$\mathbf{X}' = \sum_{k=1}^{K} \mathbf{Z}^{(k)} \cdot \mathbf{\Theta}^{(k)}$$

$$\begin{aligned} \mathbf{Z}^{(1)} &= \mathbf{X} \\ \mathbf{Z}^{(2)} &= \hat{\mathbf{L}} \cdot \mathbf{X} \\ \mathbf{Z}^{(k)} &= 2 \cdot \hat{\mathbf{L}} \cdot \mathbf{Z}^{(k-1)} - \mathbf{Z}^{(k-2)} \end{aligned} \tag{4.1}$$

Where $\mathbf{X}$ is the attribute matrix, K is the filter size, which is a hyperparameter, and $\hat{L}$ is a normalized Laplacian as described before.

GraphConv

Moris et al. [31] relates GNN to 1-dimensional Weisfeiler-Leman graph isomorphism heuristic. The method they propose uses a simple GNN model *GraphConv*. It computes the new representation of a node x_u in the following way:

$$x_u^{(t)} = x_u^{(t-1)}\mathbf{W}_1^{(t)} + \sum_{v \in N(u)} x_v^{(t-1)}\mathbf{W}_2^{(t)}$$

where $x_i^{(t)} \in \mathbb{R}^{1\times d_{\text{feat}}}$ is a representation of a node i in time t, $\mathbf{W_1}, \mathbf{W_2} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{feat}}}$ are matrices of learnable parameters with a hyperparameter d_{out}. The sum can be replaced by other functions invariant to order of neighbors e.g. mean or max. This method represents a spatial-based model which weighs all node's neighbors the same during aggregation, but the node itself is multiplied by a different weight matrix.

SageConv

Hamilton et al. [32] proposed *GraphSage*, which is a framework to generate embeddings of nodes from their neighbors. They described a general algorithm that uses an aggregation function to collect embeddings from neighbors, which are then multiplied by a weight matrix. Aggregation function should be invariant to permutations of neighbors because they do not have any natural order. They examined three different aggregation functions:

1. *Mean aggregator* takes element-wise mean of neighbours.
2. *LSTM aggregator* uses LSTM architecture to combine representations of neighbours. LSTM is not invariant to the order of neighbors because they process them as a sequence. They deal with this issue by applying the LSTM to a random permutation of the node's neighborhood.
3. *Pooling aggregator* feeds neighbor's embedding through a fully-connected neural network and aggregates the result by taking the element-wise maximum.

They concluded that LSTM and Pooling aggregators outperform others but the gain over the mean aggregator was only marginal. The resulting GraphConv layer with mean aggregation has the following form:

$$x_u^{(t)} = \mathbf{W}^{(t)} \cdot \text{mean}_{v\in\{N(u)\cup u\}} x_v^{(t-1)}$$

where $\mathbf{W} \in \mathbb{R}^{d_{\text{out}}\times d_{\text{feat}}}$ is a matrix of learnable parameters with a hyperparameter d_{out}. This method is a spatial-based model. During aggregation, it multiplies the node and its neighbors by the same weight matrix. Therefore, it has less parameters compared to GraphConv.

GCNconv

GCN [33] is a method closely related to ChebConv. It limits the convolution to $K = 1$ and approximates $\lambda_{\text{max}} = 2$, so we get:

$$\mathbf{g_\Theta} * \mathbf{x} \approx \Theta_0\mathbf{x} - \Theta_1\mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}\mathbf{x}$$

which has two sets of parameters Θ_0, Θ_1. We set $\Theta = \Theta_0 = -\Theta_1$ to get:

$$\mathbf{g_\Theta} * \mathbf{x} \approx \Theta(\mathbf{I}_N + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}})\mathbf{x}$$

Kipf et al. [33] further used normalization trick and replaced $\mathbf{I}_N + \mathbf{D}^{-\frac{1}{2}}\mathbf{A}\mathbf{D}^{-\frac{1}{2}}$ by $\hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}$ where $\hat{\mathbf{A}} = \mathbf{A} + \mathbf{I}_N$ and $\hat{\mathbf{D}}_{ii} = \sum_j \hat{\mathbf{A}}_{\mathbf{ij}}$. Finally, for a signal $\mathbf{X} \in \mathbb{R}^{N\times C}$ with C input channels and F filters we get the convolved signal as:

$$\mathbf{X}' = \hat{\mathbf{D}}^{-\frac{1}{2}}\hat{\mathbf{A}}\hat{\mathbf{D}}^{-\frac{1}{2}}\mathbf{X}\mathbf{\Theta},$$

where $\mathbf{\Theta} \in \mathbb{R}^{C\times F}$ is a matrix of learnable parameters. Authors claim that limiting $K = 1$ reduces the risk of overfitting on a local neighborhood in the graph. However, it has less parameters than ChebConv so it can loose its expressive power.

GATConv

GATConv [34] incorporates attention mechanism into the aggregation of neighbours' vectors. GraphConv and GraphSage assumes that weights of the neighbours are identical as opposed to GATConv that learns the weights. The convolution is defined as:

$$\mathbf{x}'_i = \alpha_{i,i}\mathbf{\Theta}\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\mathbf{\Theta}\mathbf{x}_j,$$

where $\mathbf{\Theta}$ are learnable parameters, $\mathbf{x_i}$ is a representation of a node i and α measures the connection strength. We compute α as:

$$\alpha_{i,j} = \frac{\exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^{\top}[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_j]\right)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}} \exp\left(\mathrm{LeakyReLU}\left(\mathbf{a}^{\top}[\mathbf{\Theta}\mathbf{x}_i \,\|\, \mathbf{\Theta}\mathbf{x}_k]\right)\right)}.$$

where $\mathbf{a}$ is a vector of learnable parameters,$\|$ denotes concatenation of vectors, LeakyReLU is an activation function similar to standard ReLu but it allows small gradients when the unit is not active.

4.3.2 Readout layer

Readout layer merges representations of all nodes into a single vector representing the whole graph. The final representation of a graph h_G has a form:

$$h_G = R(x_v^t | v \in G)$$

where $R(\cdot)$ is the readout function. Examples of readout functions are mean, max, or sum. The readout layer is necessary for graph classification tasks. We can not simply concatenate vectors of all neighbors because graphs vary in the number of nodes. Therefore, we extract a single vector representing the whole graph.

4.3.3 Multi Layer Perceptron

Multi Layer Perceptron (MLP) consists of one or more linear layers, each followed by a non-linear activation function. It is a common part of neural networks. In the case of GNN, it is used as a classifier of graph representations. MLP learns a function $f : \mathbb{R}^d \to \mathbb{R}^o$ where d is a dimension of input and o is a dimension of output.

One layer l of MLP takes the following form:

$$\mathbf{z}^{l+1} = \mathbf{W}^l \mathbf{a}^l + \mathbf{b}^l$$

$$\mathbf{a}^{l+1} = \sigma(\mathbf{z}^{l+1})$$

where $\mathbf{z} \in \mathbb{R}^d$ is an input vector. We set $\mathbf{z}^0$ equal to input features of the whole network. $\mathbf{W} \in \mathbb{R}^{o \times d}$ is a weight matrix, and $\mathbf{b} \in \mathbb{R}^o$ is a vector of biases, both are randomly initialized in the beginning and updated during the training. σ is a differentiable non-linear function. We use rectified linear unit ($ReLU$) in our work:

$$f(x) = \max(0, x)$$

ReLu is well suited for usage in neural networks because it can be computed efficiently, helps with gradient propagation in deep networks, and has many other advantages.

In our case, MLP follows the readout layer, so d in the first layer is the dimension of the vector representing the whole graph, and o in the last layer is the number of classes we are classifying.

4.3.4 Softmax Layer

Softmax is used as the last layer of neural networks to transform the output into probability-like distribution. It is defined as:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

It takes a vector $\mathbf{z}$ with a dimension equal to the number of classes being predicted and transforms it to sum up to 1. Then, this vector can be interpreted as probabilities.

4.3.5 Training a neural network

Suppose we have a training set $T = \{(x_1, y_1), \ldots, (x_m, y_m)\}$ where m is the number of samples. Each sample x_i has its corresponding label y_i. We define a cost function J called negative log likelihood:

$$J = -\sum_{i=1}^{m} \sum_{k=1}^{K} [\![y_i = k]\!] \log \hat{y}_{ik}$$

The outer sum is a sum over all training samples, K is the number of classes and $[\![\cdot]\!]$ is an indicator function. For a sample i from the class k, the loss is a logarithm of the output of the softmax $\hat{y}_{ik}$.

The loss function can be optimized using a backpropagation algorithm, which propagates the loss through the network to update its parameters accordingly. In case of a linear layer, an iteration of a *gradient descent* looks as follows:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b)$$

$$b_{i}^{(l)} = b_{i}^{(l)} - \alpha \frac{\partial}{\partial b_{i}^{(l)}} J(W, b)$$

where α is called a *learning rate*, which controls the step size when moving toward a minimum. Gradient descent is an optimization algorithm, and as we can see, we need a derivative of the loss function with respect to the parameters. The chain rule is adopted for the effective computation of derivatives. We do not give derivatives of all layers in the network because frameworks like *torch* perform automatic gradient computation. In our work, we use an extended variant of a gradient descent algorithm called Adam [35] that incorporates adaptive learning rate and momentum during the optimization. Adam is standardly used in other works to train a GNN [36].

Chapter 5

Dataset

A collection of data is necessary for machine learning algorithms to work. However, a proper dataset must comply with several requirements:

- Dataset should be large - More data helps to prevent overfitting of learned models.
- Broad-scale of samples should be present - Different samples help to learn different behaviors of files.
- Real-world scenario should be simulated - Dataset should contain data coming from the real world (not synthetic). Data should be gathered in a consecutive period to capture the evolving nature of malware.

An ideal dataset should also be publicly available to make a comparison with other approaches possible. Unfortunately, there are no public datasets suitable for our task. These are datasets broadly used in literature about static analysis:

- Microsoft challenge malware dataset [27] - more than 20,000 malware samples in the form of disassembly code and byte code. This dataset was published in 2015; thus, it can be outdated now. It does not include legitimate samples. Moreover, it does not come with function call graphs.
- Ember dataset [37] - 1 million files collected in 2018. The authors parsed the PE header of the files and extracted features out of them. Although this dataset may be large enough, it does not contain disassembled code.

We created our dataset that complies with our requirements. We provide details about data collection and labeling in the following sections.

5.1 Data collection

We collected binaries between September 1st and October 14th from a Threat-Grid (TG) [38] service. We split the data into one month of training data and

		(a) Original count	(b) Disassembled successfully	(c) Count after filtering trivial
Training set	Malicious	4,030,370	3,911,889	3,842,747
	Legitimate	599,434	483,819	312,922
Testing set	Malicious	1,664,949	1,621,996	1,601,427
	Legitimate	274,489	228,757	153,654

Table 5.1: Distribution of files in our dataset. (a) shows number of files obtained during data collection. (b) shows number of files that could be successfully disassembled. (c) shows number of remaining files after filtering out files with only a single function having only one opcode.

two weeks of testing data. This split simulates a real world scenario where we take collected samples, use them to train a model and use the model on previously unseen data. We obtained 6,569,242 files in total. Surprisingly, the dataset is heavily imbalanced towards malicious file. This fact is given by the nature of the TG service. TG is advanced tool for malware analysis so malicious files are uploaded there more often. We do not balance our dataset in any way, we rather adapt our methods to this scenario. Although, there are usually more legitimate files in real-world scenarios.

5.2 Data labelling

We obtained labels for the binaries from Reversing Labs' service called Titanium Cloud [39]. It provides File Reputation and Intelligence. They combine multiple sources to produce verdicts about files. They claim their database consists of 8 billion unique file records. All of them were processed by Active File Decomposition (AFD). AFD automatically performs several analysis techniques to extract Proactive Threat Indicators. It starts with static analysis techniques that extract underlying structures of the file by unpacking, de-archiving. Then, TC analyses network indicators, certificate indicators, known exploits, results from other AV vendors, and many other sources. Combining all their features, they are able to produce one of the three labels *malicious*,*legitimate*,*suspicious* together with family labels. We use files with *malicious* and *legitimate* labels and omit files with *suspicious* label because they do not have any clear description and could easily fool learning methods. Distribution of labels in our dataset is shown in Table 5.1.

5.3 Data analysis

In this section, we analyze our dataset to get insight into the data so we can develop suitable methods for the problem we are solving. In our analysis, we focus on information relevant to our task, i.e., disassembly and FCG.

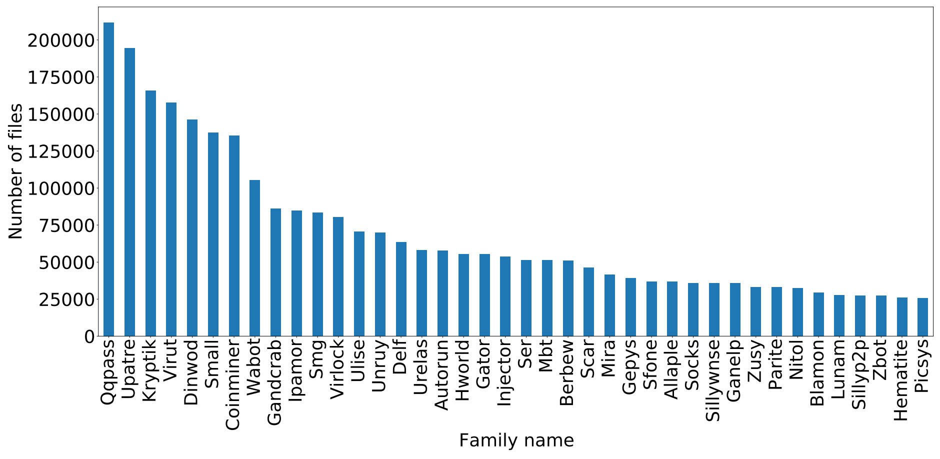


Figure 5.1: 40 largest malware families in our dataset.

5.3.1 Malware families

Malware can be divided into malware families based on its authorship or functionality. Therefore, we want to exploit the similarity between samples to detect their variants. We check the distribution of malware families in our dataset to see which ones are present and which are the most prevalent ones. We obtained family labels from the TC service that creates the labels mainly from detections of other AV vendors. Usually, reasonable names are given, but sometimes the detection is only a generic one. The 40 largest families are shown in Fig. 5.1. We can see that there are several families with more than 100,000 files, but smaller families are more common.

We give a description of some malware families based mainly on Talos [40]:

- Qqpass - malware stealing QQ messenger passwords
- Upatre - backdoor trojan downloading a malicious payload
- Kryptik - generic detection of a Windows trojan collecting system information or dropping other payloads
- Virut - widespread virus infecting other files in the computer, establishing IRC-based backdoor
- Dinwod - polymorphic dropper, obfuscated with anti static analysis tricks
- Small - downloader distributed over spam e-mails
- Coinminer - cryptocurrency miners or miner droppers

We can see that our dataset contains various malware families with diverse behavior, so it presents an interesting task for malware classification.

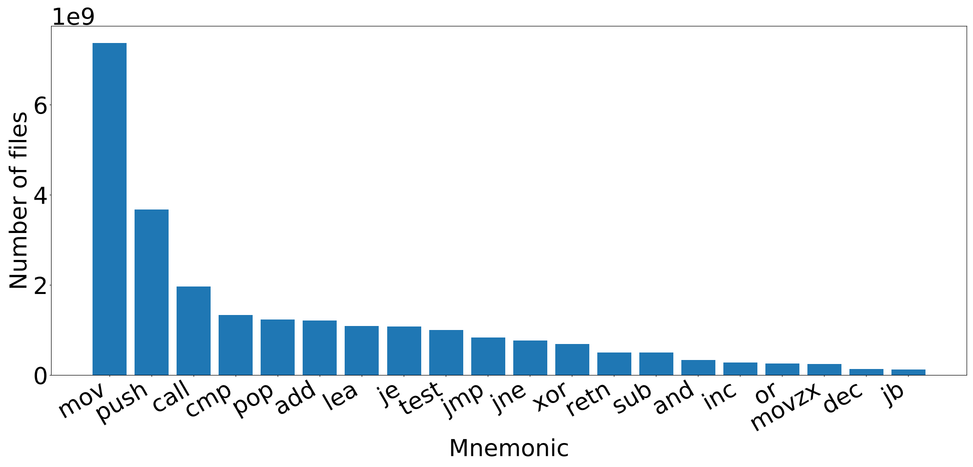


Figure 5.2: The 20 most prevalent mnemonics in our dataset.

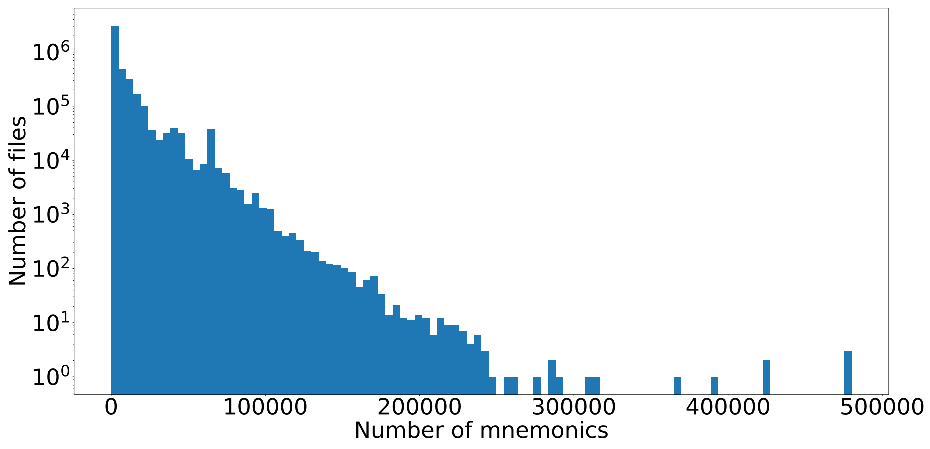


Figure 5.3: Histogram of mnemonics in our dataset. Y axis is in log scale.

5.3.2 Opcodes

By running BinaryNinja [20], we obtained a disassembly listing of each executable as described in Section 2.2.3. We were able to successfully disassemble 95% (4,395,708) of our 4,629,804 binaries in the training set and 95% (1,850,753) of 1,939,438 in the testing set. We can see that 81.5% of legitimate samples were disassembled successfully compared to 97.2% of malicious samples. Detailed numbers are in Table 5.1.

We computed the statistics of mnemonics in our training set. There were 1,139 unique mnemonics in total. The 20 most prevalent ones are shown in Fig. 5.2. We can see that the most prevalent is `mov` followed by `push` and `call`. We show a histogram of the number of instructions in a single file in Fig. 5.3. We can see that many files have only a few instructions. There are also some files with more than 400,000 opcodes.

5.3.3 Function call graphs

During our disassembly procedure, we also extracted the Function Call Graph (FCG), which represents caller-callee relations between functions within a file. Examples of FCGs of two samples from the Qqpass family are in Fig. 5.4. We can see that the shape of FCG varies significantly, even within the same malware family.

After manual analysis, we found out that in some cases, there are graphs with only a single node containing only a single instruction. These binaries usually contain only a call to a virtual machine, so no other instructions could be found. These graphs do not carry enough information for our task, so we filter them out. More samples should probably be filtered out to increase the reliability of the method, but we leave it for future work. 4,155,669 graphs remained in our training set and 1,755,081 in the testing set. We work only with these binaries in further work. Detailed numbers can be found in Table 5.1.

Graphs in our dataset are very heterogeneous. We computed the statistics of all graphs in the training set. Results are presented in Table 5.2 and we briefly discuss them:

- Number of edges in a graph - We can see that the median is 12 while mean being 258 with a large standard deviation. This indicates the presence of outliers. Truly, we can see that the largest graph has 27.581 edges.
- Number of vertices in a graph - Median is 12 and mean 112. Again, we see exceptions of very large graphs with thousands of vertices.
- Node degree - All functions (nodes) were extracted, and we computed their degrees. We can see that the median is 3, which indicates that nodes are well distributed and separated.

Extracted statistics show a great variety in our data. We would like to emphasize two cases:

- Graphs with only a few nodes but with many opcodes - These cases will examine the quality of features describing each function.
- Graphs with many functions containing only a few instructions - These samples should verify the ability to exploit graph structures.

5.4 Resulting dataset

We created two datasets in our work. The first one serves for the task of detecting malicious and legitimate files. The second one is used to solve the task of malware family classification. Unfortunately, we can not publish our datasets. However, we believe that testing our approach on data collected in the wild is valuable to the community even without publishing the data.

	mean	std	min	25%	50%	75%	max
Number of edges	258.03	572.14	0	1	12	287.00	27581
Number of vertices	112.89	218.70	1	2	12	150	10524
Node degree	4.57	7.03	0	2	3	5	3074

Table 5.2: Statistics of graphs in our dataset.

	Training set	Testing set
Qqpass	211673	69548
Upatre	193926	102551
Virut	157479	57660
Kryptik	154710	52701
Dinwod	146244	69389
Small	137387	52892
Coinminer	122828	136561
Wabot	105246	33389
Gandcrab	86109	31785
Ipamor	84304	8482

Table 5.3: Statistics of the multi-class dataset which is comprised of the 10 largest malware families in our data.

5.4.1 Two-class dataset

Two-class dataset contains all the files as detailed in Table 5.1. Each file is labeled either as malicious or legitimate. In our experiments, we use data that passed the filtering phase, i.e., disassembled successfully and with non-trivial functions. There are $\approx 10\times$ more malicious samples than the legitimate ones. The methods should be adapted to this scenario.

5.4.2 Multi-class dataset

The multi-class dataset contains samples of the 10 largest families in our dataset. These families are Coinminer, Dinwod, Gandcrab, Ipamor, Kryptik, Qqpass, Small, Upatre, Virut, Wabot. The dataset statistics are shown in Table 5.3. We see that Qqpass is the largest family in the training set, followed by Upatre and Virut. However, the distribution of the families is quite different in the testing set. Qqpass is the 3rd largest family, Upatre remains 2nd, and Virut is 5th. The largest family in the testing set is Coinminer, which was 7th in the training set. The difference in the distribution can cause troubles during the training and evaluation, so we should be aware of that.

5.5 Evaluation metrics

It is essential to measure the quality of a model/algorithm in order to start improving it. A suitable metric should reflect the real use case of the developed method. Subsequently, this metric can be directly or undirectly optimized. We measure different metrics for malware detection task and malware family classification task. We do so because there are different requirements in both cases.

5.5.1 Two-class classification metrics

In the case of malware detection, the task is:

1. Maximize the number of detected malicious files
2. Minimize the number of false alarms

We want our system to detect malicious files so we can alert a user that he/she might be infected. However, triggering alarms on legitimate files too often would result in users disabling the system. We use standard metrics to measure these requirements. We operate with terms *positive* and *negative* which are common in classification tasks. In our case, positive refers to malicious files.

True positive rate

True positive rate (TPR) (also called *recall*) measures the ratio of correctly recognized malicious samples. It is defined as:

$$\texttt{TPR} = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FN}},$$

where TP (true positives) are malicious samples classified as malicious and FN (false negatives) are malicious samples classified as legitimate.

False positive rate

False positive rate (TPR) measures the ratio of incorrectly classified legitimate samples. It is defined as:

$$\texttt{FPR} = \frac{\mathrm{FP}}{\mathrm{FP} + \mathrm{TN}},$$

where FP (false positives) are legitimate samples classified as malicious and TN (true negatives) are legitimate samples classified as legitimate.

ROC curve

Receiver operating characteristic (ROC) is a plot that shows a classifier's performance for various discrimination thresholds. The plot has FPR on its x-axis and TPR on the y-axis. The higher the curve is in the plot, the better is the model's ability to separate two classes. Many times, we compute an *Area under curve* (AUC). AUC helps us understand the model and compare multiple models [41].

5.5.2 Multi-class classification matrix

We want to categorize malicious files into malware families. Knowing the family of a file can help understand the file's behavior and speed up the subsequent analysis. Therefore, we want to classify correctly as many samples as possible.

Accuracy

Accuracy measures the ratio of correctly classified samples. It is especially useful for multi-class classification. Accuracy is defined as:

$$\texttt{ACC}(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{n} \sum_{i=0}^{n} [\![\hat{y}_i = y_i]\!]$$

where n is a number of samples, $\mathbf{y}$ is a vector of true labels, $\hat{\mathbf{y}}$ is a vector of predicted labels, and $[\![\cdot]\!]$ is an indicator function.

F1 score

F1 score is a harmonic mean of precision and recall. Firstly, let us define precision as:

$$\texttt{precision} = \frac{tp}{tp + fp},$$

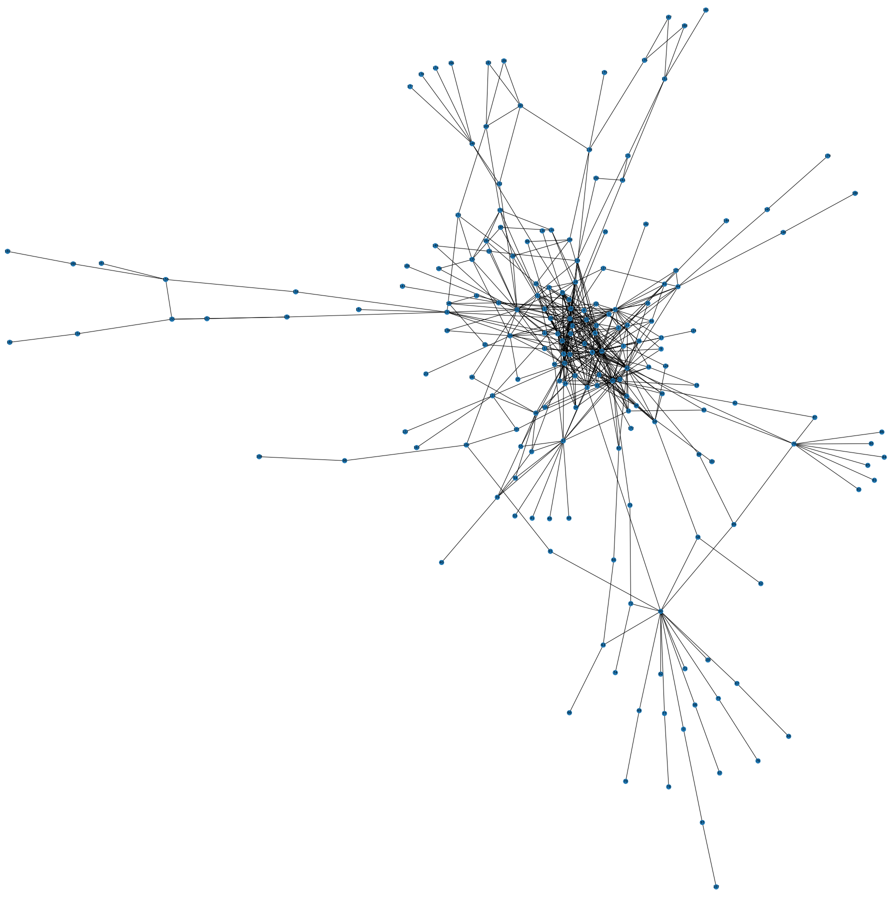
Then, F1 score is defined as:

$$F_1 = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}.$$

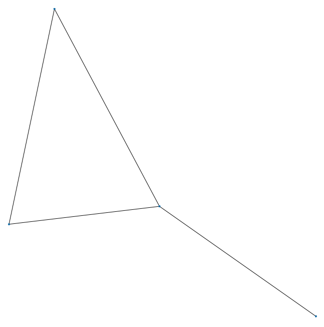
as seen in the formula the contribution of precision and recall is the same. We want to compute F1 score in multi-class classification so we must define an averaging scheme. We use weighted version of the score:

$$\frac{1}{\sum_{l \in L} |y_l|} \sum_{l \in L} |y_l| \, F_1(y_l, \hat{y}_l)$$

where L is the set of labels, $\mathbf{y}$ is a vector of true labels, $\hat{\mathbf{y}}$ is a vector of predicted labels. We want to reflect different sizes of groups of samples so we weight them by their group sizes.



(a):



(b):

Figure 5.4: FCGs of two samples from Qqpass malware family (a) More structured graph with many vertices and edges (b) Smaller graph with only a few vertices and edges.

Chapter 6

Method

We introduce several models that can be used for large scale classification of PE files. They employ the disassembled code of binaries to recognize their malicious intent. Firstly, we give details about GNN models that take advantage of FCGs to detect similarities in program structures. Secondly, we describe models that do not take the structure of an FCG into account.

In our work, we go beyond a raw bytes representation of a PE file, which can be easily obfuscated [23] [24]. Prior art proved that using assembly instructions can be successfully employed for malware classification [8] [7] [9]. We utilise FCGs together with assembly code of each function to detect structural similarities of binaries as successfully done in previous works [10] [11] [12] [13]. However, we go beyond the heuristic matching of graphs and use neural networks to do the task. We follow works which use neural networks to model code [29] more specifically graph neural networks (GNNs) as in [15] [14].

Having the GNN model, it can be compared with a model that does not take the structure of FCG into account. Comparing structure-agnostic models with GNNs was stressed in prior art [36]. By comparing the models, we can understand whether the graph's structure matter. We implemented all the models using *Pytorch geometric* (PyG) [42] framework.

6.1 GNN architecture

We want to build a GNN model that would classify PE files either as malicious/legitimate; and a second one that would classify files into according malware families. We explained a structure of GNN in Chapter 4 along with its components. We are going to adapt GNNs to extract structural information from FCGs. FCG is a graph with a set of vertices V where $v \in V$ corresponds to a function. An edge $e = (u, v)$ signifies that a function u calls a function v.

Since we want to classify the whole files, we are doing a graph classification task. The structure of our model is shown in Fig. 4.1. GNN will create a meaningful representation of each node in the graph. Afterward, we aggregate all nodes to create a single vector representing the whole graph, which is then passed to MLP to produce a label. We are going to describe which methods we implemented in each part.

6.1.1 Features

Each node (function) in a graph has a vector $\mathbf{x} \in \mathbb{R}^{d_{\text{feat}}}$ with features describing it. There is no general rule for selecting the features. We were inspired by works using instructions to perform malware classification [7]. We extracted all instructions of each function and stripped them of their operands. We wanted to decrease the number of distinct instructions like that. Then, we selected the 20 most prevalent mnemonics in the training set and used their counts as features. The selected mnemonics are:

- `mov, push, call, cmp, pop, add, lea, je, test, jmp, jne, xor, retn, sub, and, inc, or, movzx, dec, jb`

Finally, we summed occurrences of other mnemonics in the function and appended this sum to the vector. The resulting vector describing the function has a dimension $d_{\text{feat}} = 21$.

Several works suggested using node degrees as features [36] [32]. A node degree is a number of edges connected to the node. We explored this feature, as well.

We note that different features should be investigated and compared to our approach. We were more interested in getting the notion of GNNs, their behavior, and their relation to non-structural models. We see opportunities in different parsing of instructions where we completely omitted operands. We also represent the mnemonics only in a bag-of-words fashion, and more elaborate approaches could be tried [43]. We leave those challenges for future work.

6.1.2 Convolutional layer

We start by performing convolution upon the graphs. There are many possible convolutional layers [16] [17]. We use the ones explained in Section 4.3.1: ChebConv, GraphConv, SageConv, GCNConv, GATConv. A convolutional layer has a weight matrix $\mathbf{W}$ corresponding to it. These weights are learnt during the training. The dimension of the weight matrix $d_{\text{hid}} \in \mathbb{R}^{d_{\text{hid}} \times d_{\text{feat}}}$ (we call it *inner dimension*) is a hyperparameter which can be tuned. Multiple convolution layers can be stacked in order to propagate the information further in the network. By stacking the layers, we incrementally pass and aggregate the information deeper in the network. We stack up to two convolution layers to test if multiple convolutional steps improve the model. We did not go further because increasing the number of layers also increases the training time.

6.1.3 Readout layer

When the data pass through convolutional layers, we aggregate the embeddings from the whole graph to extract a single vector representing the whole graph. This vector has the same dimension as the individual embeddings i.e. d_{hid}. We described the readout layer in Section 4.3.2. We try three different functions: sum, mean, and max.

6.1.4 Classifier

We use Multi Layer Perceptron (MLP) followed by softmax as a model used to classify the graphs. We described the MLP and softmax in Section 4.3.3. We fixed the architecture of the MLP in our work in order to measure only differences in GNN architecture.The model has 3 fully connected layers each one followed by a ReLu activation function. The dimensions of the fully connected layers:

1. 1st layer $d_{\text{graph}} \times 128$
2. 2nd layer 128×64
3. 3rd layer $64 \times$ *number of classes*

6.2 Structure-agnostic model

Errica et al. [36] made an argument that a structure-agnostic model should accompany each GNN model. Moreover, several works claimed that GNN could exploit FCG [15] [14]. Therefore, we expect that including the graph structure improves the model. We introduce a new model to verify this fact. The structure-agnostic model, which we call *Linnet*, does not use the graph structure.

If the performance of a GNN model is close to this baseline, it can have two reasons:

1. The task does not need structural information to be solved.
2. GNN does not exploit the graph structure sufficiently.

A human expert can validate the first case. However, validation of a large dataset would be too time-consuming. The second case is even more difficult to asses. There can be several reasons causing it [36]. Starting by the bias induced by the architecture of the network and its hyperparameters. Another significant factor is the amount and distribution of the training data.

The model has the same structure as the GNN model except for the graph convolutional layer. The Linnet model is shown in Fig. 6.1. The convolution is replaced by a linear layer mapping the input features to the higher dimension d_{hid}. A ReLU activation function follows the linear layer. These embeddings

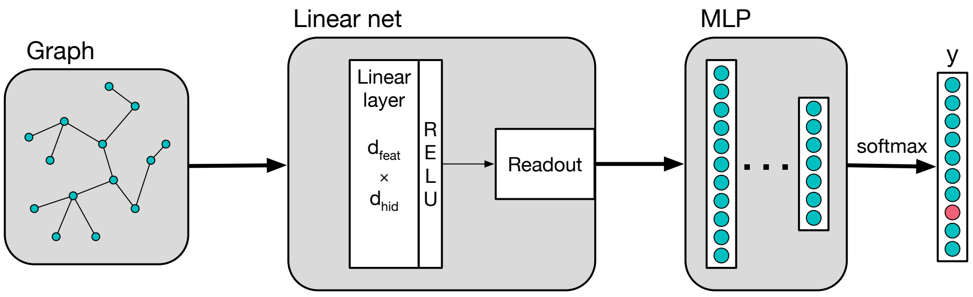


Figure 6.1: Linnet model structure.

in the higher dimension are then aggregated in the readout layer. Finally, graph embedding is fed to MLP, and a class label is produced. Obviously, edges in the FCG are not respected at all.

Chapter 7

Results

Since GNN has started developing quite recently and only a few works adapted them to malware classification, we compare several GNNs and study effects of tweaking their parameters. We start by comparing different kinds of convolutional layers. There have been many works published [17] [16], but only a few have been tested on the malware classification task. Therefore, we study the impact of each type on the classification performance. We also asses different sizes of hidden layers in the network. We go further and compare different ways of aggregating features from a node's neighborhood as well as from the whole graph. Finally, we compare our best GNN model with the structure-agnostic model to see whether the graph structure matters. We believe these experiments will serve as an excellent introduction to GNNs and will shed some light on their adoption for malware classification.

We performed our experiments on two classification tasks. The first one is a binary classification where we classify a PE file either as malicious or legitimate. The second task is a multi-class classification where we classify malicious files into 10 malware families. All models were implemented using Pytorch geometric (PyG) [42] and trained on AWS machine r5a.16xlarge. We use default parameters of PyG 1.4.3 if not stated otherwise. We fix the learning rate at 10^{-4} across the experiments because it ensured the smooth convergence of the models compared to different values.

All the experiments are performed on the dataset that we introduced in Chapter 5. We additionally created two validation sets:

1. Binary classification validation set - we randomly select 20% of the training set and use it as a validation set.
2. Multi-class classification validation set - we randomly select 10% samples from each malware family in the training set and use it as a validation set.

The validation set represents unseen data during the training, so we use it to select the best model after the training. We simply choose the model with the lowest error on the validation set.

We run all experiments under these circumstances:

- We use the same data, and the same data splits (training, validation, testing set)
- Each model was trained for 20 epochs for binary classification and 30 epochs for multi-class classification. After those epochs, we did not see much improvement.
- The architecture of the neural network is fixed except switching the convolutional layer.
- If not stated otherwise, the dimension of the weight matrix in the convolutional layer was set to 128.
- If not stated otherwise, we concatenate the output of max and mean readout creating a new representation of the graph where $d_{\text{graph}} = 256$.
- The data were fed to the network in batches of size 512.
- In case of binary classification, the weight of legitimate samples was set to 10 because we have $\approx 10\times$ more malicious samples than legitimate ones

7.1 Convolutional layers comparison

Many different kinds of convolutional layers have been published; thus, newcomers might get lost in them pretty easily. Therefore, we empirically compare several works to see their impact on classification performance.

7.1.1 Multi-class classification

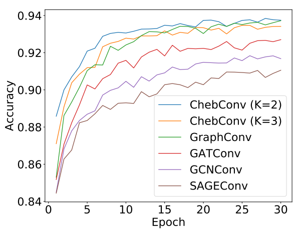
We compare ChebConv, GraphConv, SAGEConv, GCNConv, and GATConv in this experiment. The progress of loss function, accuracy, f1 score during the training is shown in Fig. 7.1. The name of a model corresponds to the convolutional layer used. We can see that the loss is steadily decreasing, and accuracy and f1 scores are rising in all cases. GATConv, GCNConv, and SAGEConv are performing noticeably worse than other models.

We select a model for each method based on the minimal validation loss. The results of all the models are shown in Table 7.1. We can see that the model using GraphConv has the best performance on the testing set: Accuracy 0.9416 and F1 score 0.9421.

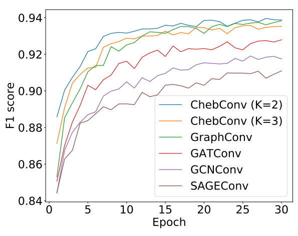
GraphConv is very similar to SAGEConv but uses two weight matrices instead of one. The first one is used to transform weights of each particular node and the second one to transform vectors of neighbors before aggregation. Having two weight matrices seems to give GraphConv a significant performance gain. Another difference is that SAGEConv computes the mean of neighbors and the current node, compared to GraphConv, which computes the sum of vectors. Later, we show that the GraphConv model with mean aggregation still has much better performance than GraphSage.

We see that ChebConv models, both with the K parameter set to 2 and 3 perform similarly, and both have a similar performance as GraphConv. We see that ChebConv (K=2) is performing the best out of all models on the training and validation set. However, it probably slightly overfitted the training set because the performance on the testing set is not increasing much after the 14th epoch. GraphConv ends up having better performance on the testing set. ChebConv (K=2 or K=3) also outperformed GCNConv, which is their simpler version. It seems that having more expressive power gives an advantage to ChebConv over GCNConv.

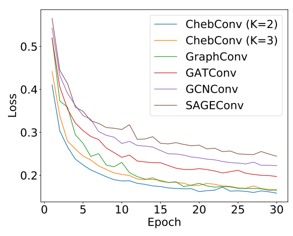
Bringing attention to the aggregation function does not help the performance. GATConv still does not match other methods that assign the same weight to each connection with a neighbor. This particular method can suffer from broad diversity in graphs where node degrees vary significantly.



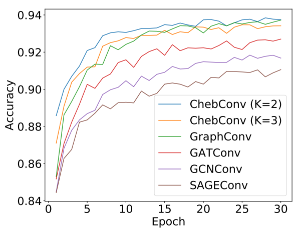
(a) : Training accuracy



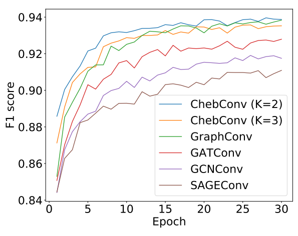
(b) : Training F1



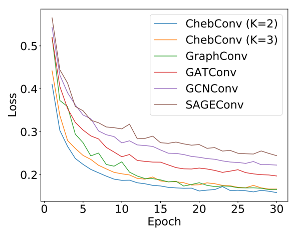
(c) : Training loss



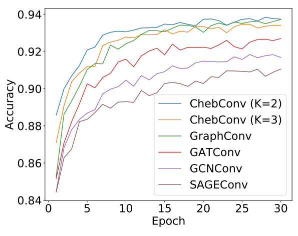
(d) : Validation accuracy



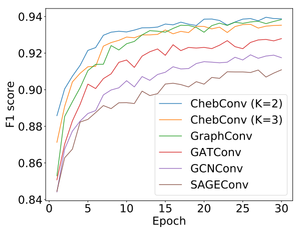
(e) : Validation F1



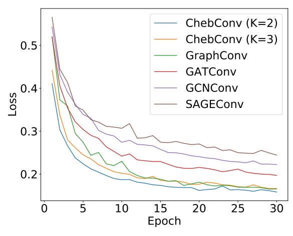
(f) : Validation loss



(g) : Testing accuracy



(h) : Testing F1



(i) : Testing loss

Figure 7.1: Comparison of various convolutional layers on multi-class classification task. The evolution of each metric during training is shown.

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
ChebConv (K=2)	**0.9379**	**0.9375**	0.9325	**0.9392**	**0.9387**	0.9328
ChebConv (K=3)	0.9359	0.9342	0.9323	0.9370	0.9353	0.9323
GraphConv	0.9358	0.9361	**0.9416**	0.9370	0.9373	**0.9421**
GATConv	0.9058	0.9270	0.9060	0.9067	0.9279	0.9058
GCNConv	0.9001	0.9168	0.8973	0.9009	0.9175	0.8966
SAGEConv	0.8889	0.9106	0.8846	0.8893	0.9109	0.8834

Table 7.1: Comparison of various convolutional layers on multi-class classification task. The best performing model on each split is in bold.

The best model analysis

We have found out that the best performing model with a single convolutional layer is *GraphConv* model. We show its confusion matrices in Fig. 7.2. We see some cases when a large portion of a malware family is misclassified, and we further looked into that.

Many samples are mistakenly labeled as Dinwod family, e.g., Qqpass, Upatre, and Kryptik samples. We analyzed these misclassified samples and found out that many of them have exactly the same graph structure and features. After digging deeper into those samples, we found out that most of them were packed by the Petite packer. We are very likely analyzing only the unpacking stub, which is the same in all the cases. We would need to perform unpacking to differentiate these samples. In our case, most of those samples are labeled as Dinwod in the training set, so they all get classified as Dinwod.

We have seen that many samples get misclassified because of packing. We do not perform unpacking in this work because it is not a trivial task, and we leave it for future work. Our approach seems to detect packers right so that it could be possibly adapted for the task of packer detection.

Motivated by the packed binaries, we analyzed how many unique samples we have in our dataset. We analyzed the validation set because it is small enough to perform a unique operation. We concatenated all edges and features in the graphs, and we found out:

- There are 19,164 graphs out of 139,994 (13.69%), which have a unique set of edges and features.
- There are 18,597 graphs out of 139,994 (13.28%), which have a unique set of features.
- There are 14,233 graphs out of 139,994 (10.17%), which have a unique set of edges.

It is only a rough estimate of the unique number of samples, but it still provides an interesting insight into the data. We see that there are only

	TPR			FPR		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv	0.9758	0.9751	**0.9676**	0.0447	0.0459	0.0530
ChebConv	**0.9767**	**0.9759**	0.9671	**0.0312**	**0.0336**	**0.0390**

Table 7.2: Comparison of various convolutional layers on binary classification task. The best performing model on each split is in bold.

13.69% unique samples based on the validation set. There are even less unique graph structures, which brings the question of whether the structure itself carries any information. The duplicate samples are spread across multiple families, as shown previously. The classifier can get stuck optimizing these samples during the training even though it obviously cannot differentiate between them. This problem can be solved by unpacking the samples or, in some cases, by creating more elaborate feature representations of nodes.

7.1.2 Binary classification

We have seen that ChebConv (K=2) and GraphConv were the best performing models for multi-class classification task. We trained the same models also for a binary classification task where we classify a file either as malicious or legitimate. The results are shown in Table 7.2. Both models have similar performance and are able to effectively recognize malicious files. GraphConv model has a TPR 0.9676 with FPR 0.053 on the testing set. ChebConv model has TPR 0.9671 with FPR 0.039 on the testing set.

7.2 Hidden layer size comparison

We tried changing the dimension of the weight matrix used in the convolutional layer. Increasing the dimension brings more learnable parameters into the network. More parameters result in a more complex model that can be susceptible to overfitting. We compared models with GraphConv and ChebConv convolutional layers because they proved to perform the best. The results are shown in Table 7.3.

We can see that ChebConv, with dimensionality 256, is the best performing model on the testing set. It reaches an accuracy of 0.9462 with an F1 score of 0.9468. Decreasing the dimensionality to 128 more or less preserves the performance on the training and validation set but worsens results on the testing set to an accuracy of 0.9325 and F1 score 0.9328. However, we see that decreasing the dimension from 128 to 32 counter-intuitively helps the performance on the testing set, which suggests that the differences are more likely due to random initialization effects.

GraphConv has the best performance on the testing set for the inner dimension 128. Increasing or decreasing the dimension slightly improves performance on the training and validation set, but it drops on the testing set.

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
ChebConv (128)	**0.9379**	0.9375	0.9325	**0.9392**	0.9387	0.9328
ChebConv (256)	0.9374	0.9367	**0.9462**	0.9385	0.9379	**0.9468**
ChebConv (32)	0.9355	0.9358	0.9409	0.9367	0.9370	0.9413
GraphConv (128)	0.9358	0.9361	0.9416	0.9370	0.9373	0.9421
GraphConv (256)	**0.9379**	**0.9384**	0.9353	0.9390	**0.9396**	0.9352
GraphConv (32)	0.9361	0.9365	0.9322	0.9374	0.9377	0.9324

Table 7.3: Comparison of various dimensions of the hidden layer in various convolutional layers on multi-class classification task. The value in brackets denotes the dimension. The best performing model on each split is in bold.

It is worth mentioning that although increasing the dimension of weight matrices can improve the models, it also notably increases the training time.

7.3 Stacking multiple convolutional layers

In the GNN model, we can stack multiple convolutional layers on top of each other to propagate information deeper in the network. Adding a second convolution will give each node information from neighbors of the neighbors. We wanted to verify if more information will help the classifier to make better decisions. We trained three models in this experiment:

1. ChebConv with two consecutive hidden layers of dimension 32
2. GraphConv with two consecutive hidden layers of dimension 128
3. SageConv with two consecutive hidden layers of dimension 128

We compare these models, with two convolutional layers, with models having only one. The comparison is provided in Table 7.4. ChebConv model provides a good performance considering we used dimension only 32 of the weight matrix. Adding one more convolutional layer improved the model's performance from the accuracy of 0.9409 to 0.9469 and an F1 score of 0.9413 to 0.9474. In the GraphConv model, we see that the results improved on the training and validation data but got worse on the testing data. This fact might indicate that overfitting occurs. In the SageConv model, the performance got significantly worse even on the training set - accuracy dropped from 0.8889 to 0.8586. Other than that, the results indicate that stacking more layers can be beneficial, but we need to mind overfitting.

7.4 Aggregation function comparison

When we defined spatial-based ConvGNNs, we said that they perform aggregation of neighbors' vectors in each node. Several possible aggregation functions can be used. The function needs to be invariant to the order of neighbors.

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
ChebConv (1 conv. layer)	0.9355	0.9358	0.9409	0.9367	0.9370	0.9413
ChebConv (2 conv. layers)	0.9366	0.9377	0.9469	0.9378	0.9388	0.9474
GraphConv (1 conv. layer)	0.9358	0.9361	0.9416	0.9370	0.9373	0.9421
GraphConv (2 conv. layer)	0.9366	0.9378	0.9381	0.9378	0.9390	0.9383
SageConv (1 conv. layer)	0.8889	0.9106	0.8846	0.8893	0.9109	0.8834
SageConv (2 conv. layer)	0.8586	0.9012	0.8517	0.8587	0.9014	0.8505

Table 7.4: Comparison of models with two convolutional layers with their simple counterparts on multi-class classification task.

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv (sum aggr.)	0.9358	0.9361	**0.9416**	0.9370	0.9373	**0.9421**
GraphConv (max aggr.)	0.9361	0.9372	0.9310	0.9372	0.9383	0.9312
GraphConv (mean aggr.)	**0.9372**	**0.9374**	0.9389	**0.9384**	**0.9385**	0.9390

Table 7.5: Comparison of *GraphConv* models with different aggregation functions on multi-class classification task. The aggregation function used is in brackets. The best performing model on each split is in bold.

We study whether changing the aggregation function has an impact on the classification performance. We performed our experiment on a GraphConv model with a single convolutional layer having the inner dimension of 128. We tried three possible aggregation functions: sum (default in PyG), mean, max. These three models differ only in the chosen aggregation function; otherwise, they have the same architecture. The results are shown in Table 7.5.

GraphConv model with sum aggregation performs the best on the testing set. It has an accuracy of 0.9416 compared to 0.9389 in the mean model and 0.9310 in the max model. The model with mean aggregation function has higher accuracy and F1 score on the training and validation set, but it has lower performance on the testing data. The max model has very similar performance as the mean model on the training and validation data, but it has lower results on the testing data. To sum up, we can conclude that using the sum aggregation is the best choice among others.

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv (sum readout)	0.9300	0.9313	0.9237	0.9312	0.9324	0.9233
GraphConv (max readout)	**0.9342**	0.9344	0.9282	**0.9354**	0.9355	0.9280
GraphConv (mean readout)	0.9324	**0.9345**	**0.9381**	0.9335	**0.9356**	**0.9385**

Table 7.6: Comparison of *GraphConv* models with different readout layers on multi-class classification task. The readout function used is in brackets. The best performing model on each split is in bold.

7.5 Readout comparison

We explained that a readout layer is necessary for the GNN when performing graph classification. Before the layer, each node has its own vector representing it. It is necessary to combine these nodes' vectors to create a single vector representing the whole graph, which is exactly done in the readout layer.

In this work, we compare the simple yet effective approaches: mean, sum, and max readout. We used the GraphConv model with an inner dimension of 128 and alternated only the readout layer. The results are shown in Table 7.6. We see that the models have similar results, but mean and max readout perform slightly better. The model with mean readout attains the best accuracy 0.9381 and F1 score 0.9385 on the testing set. The model with max readout has an accuracy of 0.9282 and an F1 score of 0.9280. We used concatenation of sum and max readout in other models. This experiment confirms that it is a reasonable choice.

7.6 Comparison with a structure-agnostic model

Linnet is a structure-agnostic model, for details see Section 6.2. We compare this model with the GNN model to check if the structure of FCG helps to make better predictions. In the *Linnet* model, we set the dimension of the weight matrix to 128, which is the same as the inner dimension in the GraphConv model.

7.6.1 Multi-class classification

Firstly, we compare models on the multi-class classification task. We show the training progress in Fig. 7.3 to see how much the models differ in convergence. The Linnet model has consistently better performance than the GNN model except for a few peaks.

We compare the performance of the models in Table 7.7. We can see that Linnet outperforms the GNN model on all the splits. It is surprising because adding structure to the model does not seem to bring any advantage, and

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv	0.9358	0.9361	0.9416	0.9370	0.9373	0.9421
Linnet	**0.9396**	**0.9398**	**0.9501**	**0.9408**	**0.9410**	**0.9507**

Table 7.7: Comparison of *GraphConv* model with *Linnet* on multi-class classification task. The best performing model on each split is in bold.

the GNN model is performing worse. We believe it is due to the combination of these factors:

- There is not enough variance in the graph structures. We have found out that only a small portion of the data has a unique structure of the graph.
- The dataset contains very similar samples (or polymorphic malware). There are large clusters of data in the dataset that can be detected just by using simple features. Then, adding the structure confuses the classifier.
- More advanced techniques to build a GNN should be used. We built only a simple GNN, which can be further improved.

Our findings are similar to the work of Errica et al. [36]. They found out that GNNs are outperformed by structural-agnostic models on several datasets standardly used for graph classification.

We further plot the confusion matrix for the Linnet model in Fig. 7.4. We can see that the largest difference between models is in the Gandcrab family, which is often confused in the GraphConv model. These samples should be further analyzed to see why the GNN model is falling behind.

7.6.2 Binary classification

We compare ChebConv and GraphConv with the Linnet model on the binary classification task. The results are shown in Table 7.8. We see that Linnet has TPR 0.9668, which is similar to ChebConv having 0.9671 and GraphConv with 0.9676. However, Linnet reaches a lower FPR of 0.0309 compared to ChebConv with 0.0390 and GraphConv with 0.0530. We plot testing set ROC curve in Fig. 7.5. We can see that the performance of the Linnet model is slightly above the other two models. Linnet model has AUC 0.9945 compared to ChebConv with 0.9935 and GraphConv with 0.9916. Again, we see that the Linnet model outperforms the GNN models.

7.7 Node-degree model

Motivated by the results in the previous section, we created a model that would exploit only the graph structure, ignoring the opcode features. Each

	TPR			FPR		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv	0.9758	0.9751	**0.9676**	0.0447	0.0459	0.0530
ChebConv	**0.9767**	**0.9759**	0.9671	0.0312	0.0336	0.0390
Linnet	0.9748	0.9738	0.9668	**0.0210**	**0.0242**	**0.0309**

Table 7.8: Comparison of GNN models with *Linnet* on binary classification task. The best performing model on each split is in bold.

	Accuracy			F1 score		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv	0.9358	0.9361	0.9416	0.9370	0.9373	0.9421
Linnet	**0.9396**	**0.9398**	**0.9501**	**0.9408**	**0.9410**	**0.9507**
Node-degree GNN	0.7246	0.7264	0.7420	0.7142	0.7156	0.7423

Table 7.9: Comparison of *GraphConv* model with baselines on multi-class classification task. The best performing model on each split is in bold.

node is described by a single number, which is the node degree. It is a recommended approach for featureless graphs [36]. The results are shown in Table 7.9. We see that Node-degree GNN has an inferior performance. It has an accuracy of 0.7420 on the testing set compared to 0.9416 of the GraphConv model and 0.9501 of the Linnet model. Therefore, it again indicates that the graph's structure itself does not carry much useful information.

7.8 Training time comparison

We have shown how different parameters affect the classification performance of GNNs. Stacking multiple convolutional layers or increasing the inner dimensions of the layers can improve the performance. However, more parameters may require more epoch passes to converge. Then, training time may be of the essence, e.g., because of time constraints. We also want to compare the cost of training Linnet and GNN models.

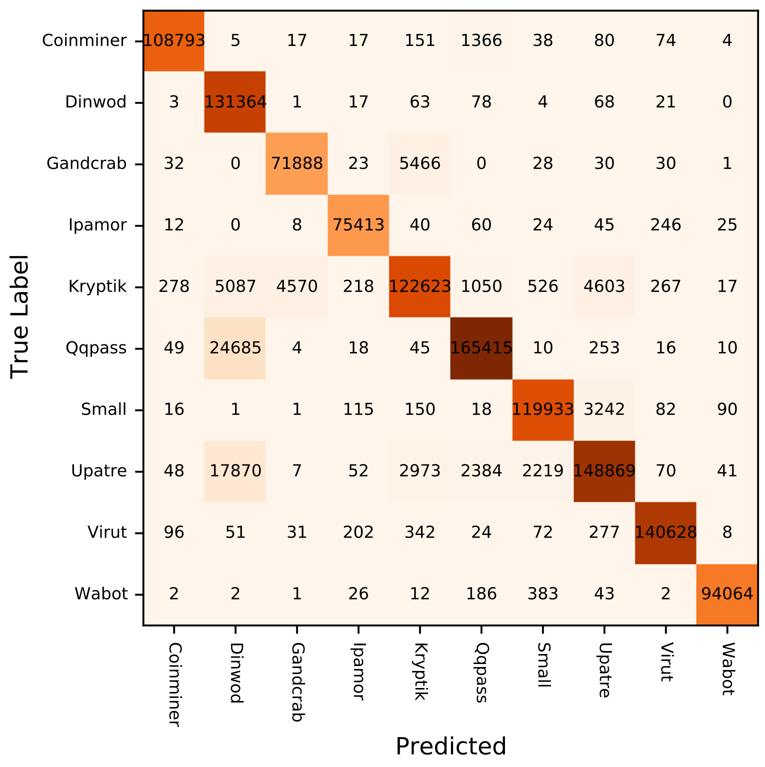
We measured the mean time needed to train for a single epoch. We show the times in Fig. 7.6. We see that generally, ChebConv is faster than GraphConv. Increasing the inner dimension from 128 to 256 increases the training time for GraphConv by $\approx 52\%$, and for ChebConv by $\approx 18\%$. Setting $K = 3$ in ChebConv slows the training time by $\approx 18\%$ compared to $K = 2$. Stacking two convolutional layers increases the training time by $\approx 116\%$ for ChebConv, $\approx 94\%$ for SageConv, but only $\approx 49\%$ for GraphConv. Linnet model is faster than all the GNN models with the same dimension and provides superior performance at the same time.

	TPR			FPR		
	Training	Validation	Testing	Training	Validation	Testing
GraphConv	0.9758	0.9751	0.9676	0.0447	0.0459	0.0530
ChebConv	0.9767	0.9759	0.9671	0.0312	0.0336	0.0390
Linnet	0.9748	0.9738	0.9668	0.0210	0.0242	0.0309
PE-GBM	**0.9973**	**0.9973**	**0.9953**	**0.0086**	**0.0080**	**0.0193**

Table 7.10: Comparison of GNN models, *Linnet* with PE-GBM model on binary classification task. The best performing model on each split is in bold.

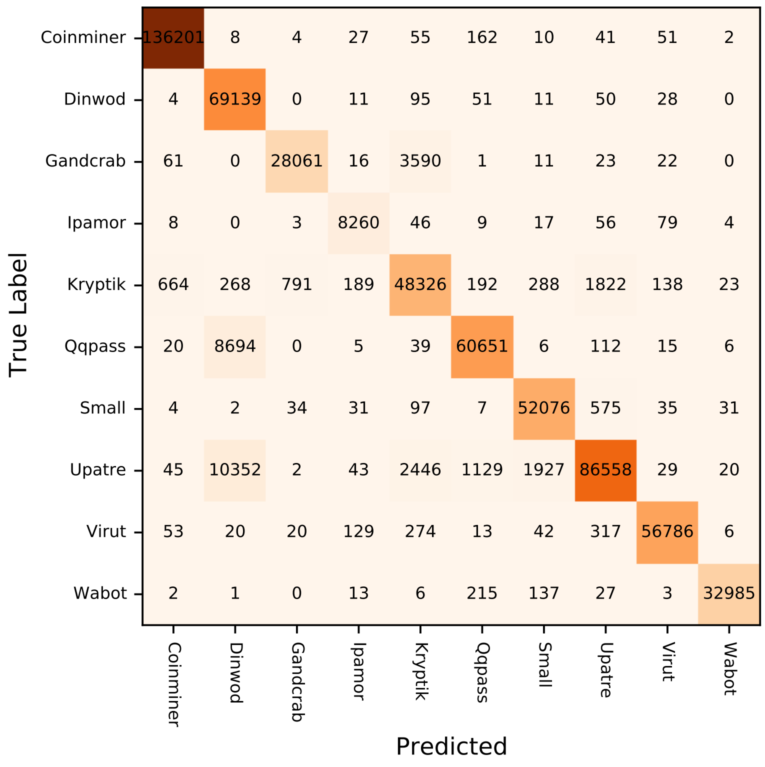
7.9 Comparison with other models

The models tested in this work provided pretty satisfying results despite using only simple features. We wanted to compare them with a different model to see where they stand globally. We used Cisco's internal static analysis engine (we call it *PE-GBM* here), which uses various features extracted from PE header and strings. It is similar to a model described in [18]. The model was trained for the binary classification task using the same training set as in the rest of this work. The results are shown in Table 7.10. We see that PE-GBM outperforms all the models. The main reason is that PE-GBM has much more features that help to make better decisions. Features extracted from PE header and strings seem to be very useful features. They should be included in a model for real deployment because they can significantly improve the model. Their adoption in GNN models can be an interesting direction of research.



True Label \ Predicted	Coinminer	Dinwod	Gandcrab	Ipamor	Kryptik	Qqpass	Small	Upatre	Virut	Wabot
Coinminer	108793	5	17	17	151	1366	38	80	74	4
Dinwod	3	131364	1	17	63	78	4	68	21	0
Gandcrab	32	0	71888	23	5466	0	28	30	30	1
Ipamor	12	0	8	75413	40	60	24	45	246	25
Kryptik	278	5087	4570	218	122623	1050	526	4603	267	17
Qqpass	49	24685	4	18	45	165415	10	253	16	10
Small	16	1	1	115	150	18	119933	3242	82	90
Upatre	48	17870	7	52	2973	2384	2219	148869	70	41
Virut	96	51	31	202	342	24	72	277	140628	8
Wabot	2	2	1	26	12	186	383	43	2	94064

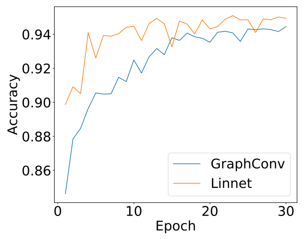
(a) : Training confusion matrix



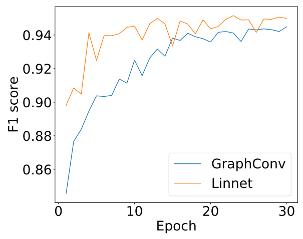
True Label \ Predicted	Coinminer	Dinwod	Gandcrab	Ipamor	Kryptik	Qqpass	Small	Upatre	Virut	Wabot
Coinminer	136201	8	4	27	55	162	10	41	51	2
Dinwod	4	69139	0	11	95	51	11	50	28	0
Gandcrab	61	0	28061	16	3590	1	11	23	22	0
Ipamor	8	0	3	8260	46	9	17	56	79	4
Kryptik	664	268	791	189	48326	192	288	1822	138	23
Qqpass	20	8694	0	5	39	60651	6	112	15	6
Small	4	2	34	31	97	7	52076	575	35	31
Upatre	45	10352	2	43	2446	1129	1927	86558	29	20
Virut	53	20	20	129	274	13	42	317	56786	6
Wabot	2	1	0	13	6	215	137	27	3	32985

(b) : Testing confusion matrix

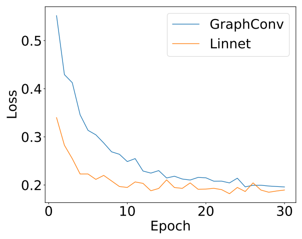
Figure 7.2: Confusion matrices for *GraphConv* model which is the best performing GNN model with a single convolutional layer of dimension 128.



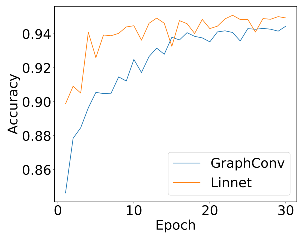
(a) : Training accuracy



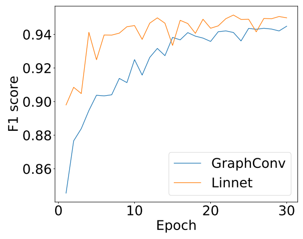
(b) : Training F1



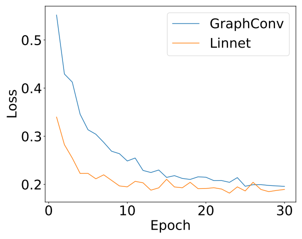
(c) : Training loss



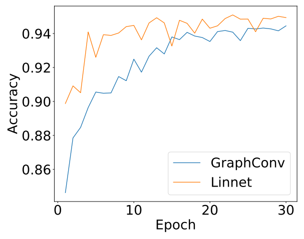
(d) : Validation accuracy



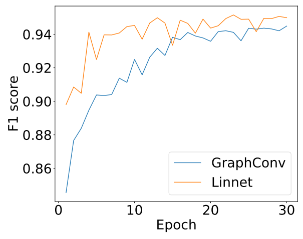
(e) : Validation F1



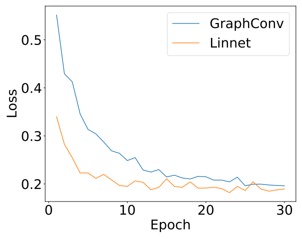
(f) : Validation loss



(g) : Testing accuracy

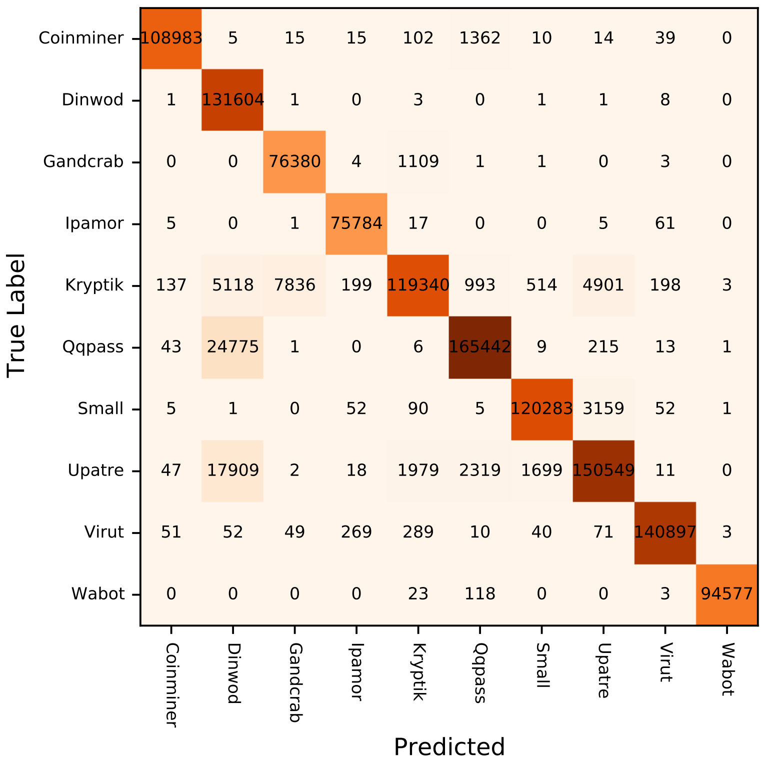


(h) : Testing F1



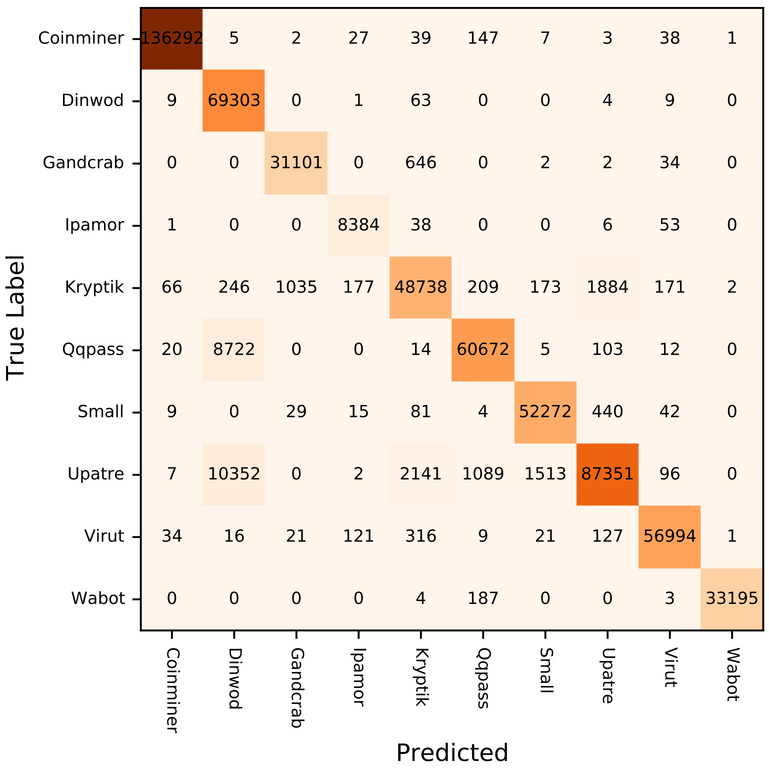
(i) : Testing loss

Figure 7.3: Comparison of the GNN model with Linnet on multi-class classification task. The evolution of each metric during training is shown.



True Label \ Predicted	Coinminer	Dinwod	Gandcrab	Ipamor	Kryptik	Qqpass	Small	Upatre	Virut	Wabot
Coinminer	108983	5	15	15	102	1362	10	14	39	0
Dinwod	1	131604	1	0	3	0	1	1	8	0
Gandcrab	0	0	76380	4	1109	1	1	0	3	0
Ipamor	5	0	1	75784	17	0	0	5	61	0
Kryptik	137	5118	7836	199	119340	993	514	4901	198	3
Qqpass	43	24775	1	0	6	165442	9	215	13	1
Small	5	1	0	52	90	5	120283	3159	52	1
Upatre	47	17909	2	18	1979	2319	1699	150549	11	0
Virut	51	52	49	269	289	10	40	71	140897	3
Wabot	0	0	0	0	23	118	0	0	3	94577

(a) : Training confusion matrix



True Label \ Predicted	Coinminer	Dinwod	Gandcrab	Ipamor	Kryptik	Qqpass	Small	Upatre	Virut	Wabot
Coinminer	136292	5	2	27	39	147	7	3	38	1
Dinwod	9	69303	0	1	63	0	0	4	9	0
Gandcrab	0	0	31101	0	646	0	2	2	34	0
Ipamor	1	0	0	8384	38	0	0	6	53	0
Kryptik	66	246	1035	177	48738	209	173	1884	171	2
Qqpass	20	8722	0	0	14	60672	5	103	12	0
Small	9	0	29	15	81	4	52272	440	42	0
Upatre	7	10352	0	2	2141	1089	1513	87351	96	0
Virut	34	16	21	121	316	9	21	127	56994	1
Wabot	0	0	0	0	4	187	0	0	3	33195

(b) : Testing confusion matrix

Figure 7.4: Confusion matrices for *Linnet* model which is a structure-agnostic model.

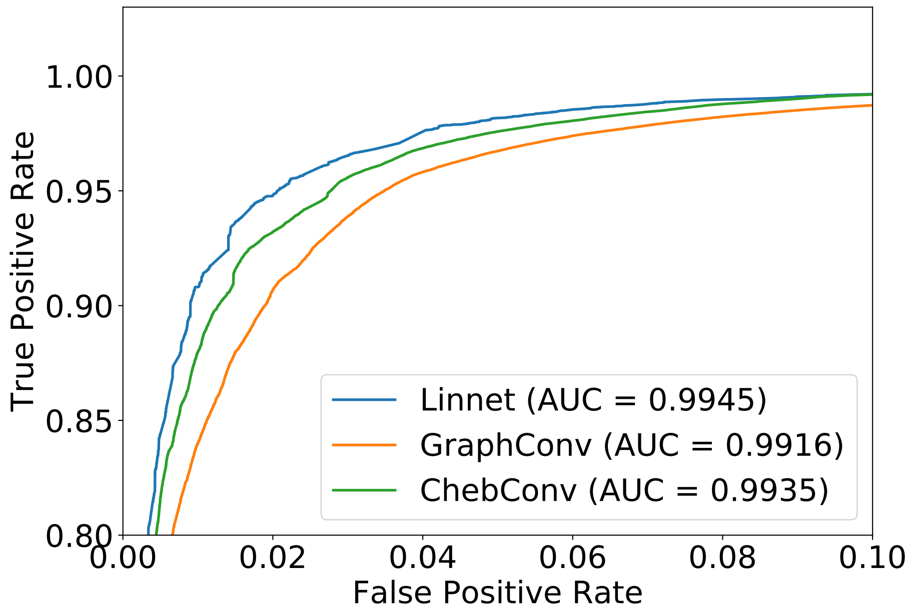


Figure 7.5: ROC curves of GNN models and Linnet on binary classification task. Zoomed in at top left corner.

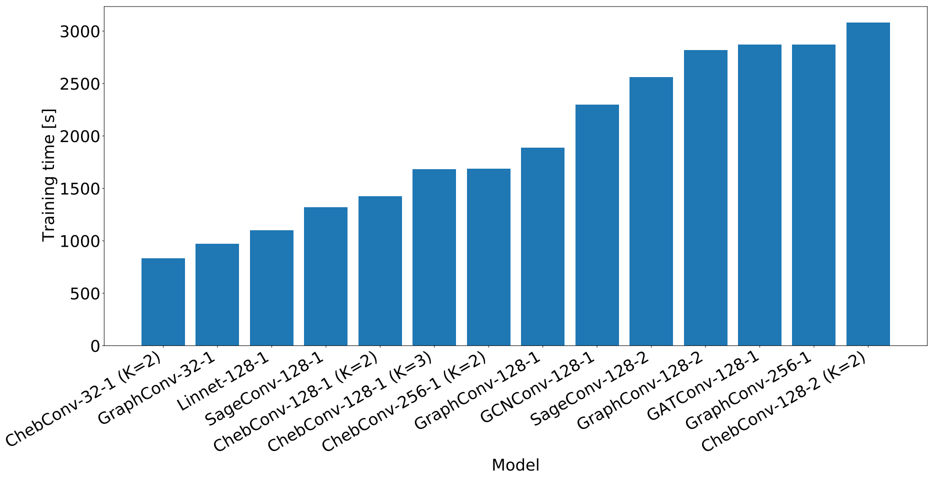


Figure 7.6: Comparison of time needed for an epoch of training. The format of name is *{method name}-{inner dimension}-{number of conv. layers}*.

Chapter 8

Conclusion

In this thesis, we studied Graph Neural Networks (GNNs) and their usability on the malware classification task of PE files. We described how a GNN model could be built to work on Function Call Graphs (FCGs) and overviewed different components that compose it. All our experiments were performed on a large dataset of more than 5 million PE files, which is much larger than the ones commonly used in the literature.

We compared five types of GNN convolutional layers and found out that *GraphConv* and *ChebConv* have the best performance out of them. Other models had significantly worse results. The weighting of neighbors during the aggregation step in *GATConv* did not bring any improvements in models. This particular method can suffer from broad diversity in graphs where node degrees vary significantly. *GCNConv* model is very similar to *ChebConv*, but it is more shallow. The results showed that more expressive power in *ChebConv* helps the performance. *SAGEConv* is considered a good baseline model for graph classification tasks. However, it had the lowest performance in our experiments. It is a similar model to *GraphConv*, which uses one weight matrix for a current node and a different one for its neighbors. *GraphConv's* weighting scheme evidently leads to better results.

We further found out that stacking multiple *ChebConv* convolutional layers is beneficial for the performance. Increasing the dimensions of weight matrices in convolutional layers improved again only the *ChebConv* model. The same happened when we decreased the dimension, which suggests that the differences are more likely due to random initialization effects.

Having studied different kinds of GNNs, we were able to select the best performing GNN models on our data. Finally, the GNN models were compared to a *Linnet* model - a model that does not take the FCG structure into account. We expected this model to have lower performance because it performs only a simple aggregation of nodes' representation. It does not propagate information to neighbors over FCG's edges as in GNN. Nevertheless, the structure-agnostic model outperforms the GNN models on both binary and multi-class classification problems.

In the case of multi-class classification, the best performing GNN models with a single convolutional layer are *GraphConv* with an accuracy of 0.9416 and weighted F1 score 0.9421; and *ChebConv* with an accuracy of 0.9462 and weighted F1 score 0.9468. The *Linnet* model has an accuracy of 0.9501 and a weighted F1 score of 0.9507.

From our experiments, we conclude that the graph structure of the PE file does not bring advantages regarding classification performance. GNN models we tried are not able to exploit the graph's structure yet. It points to the fact that the PE files classification might be done reasonably accurately without employing the structure; or other effects like data distribution and model's architecture can cause the error as well. In our particular case, we found out that FCGs in our dataset did not contain enough variance, and most of them were identical. Additionally, a model using only the graph structure with node degrees as features exhibited poor results. The FCGs do not seem to carry enough structural information. Interestingly, other works observed the same effect on other datasets standardly used in graph classification research - the structure-agnostic models outperformed GNNs as well [36]. Obviously, it is not only a problem with the malware classification task, so researchers should focus on explaining this phenomenon.

Furthermore, we analyzed our results and found out that the packing of binaries is a massive obstacle for our method and, consequently, to other static analysis methods. Packed binaries exhibited the same structures of FCGs, so the analysis methods cannot differentiate between them.

Comparing the previous models with an industrial static analysis engine showed that the data contain more information that can be utilized. It was able further to improve the TPR by $\approx 3\%$ while decreasing FPR. The model did not perform unpacking of the files but only used a different set of features, especially those extracted from the PE header. Our GNN models used very simple features, so more elaborate ones should be included in the model to obtain better results.

In future work, the FCG structure's contribution should be further investigated to explain the phenomenon of the lower performance of GNN models when compared to non-structural models. GNN models should be tested on different datasets and carefully analyzed whether they exploit the graph structure at all. Secondly, the FCG should be enriched by more sophisticated features describing nodes. Thirdly, time should be invested in the unpacking of samples. It would significantly increase the reliability of the method. Additionally, methods for explaining GNN have been published [44]. They can be used to analyze the nets and also help to identify important functions in the FCG.

Appendix A

Bibliography

[1] AV-TEST. Av-test - independent it-security institute. `https://www.av-test.org/en/`. [Online; accessed 13-March-2020].

[2] OSDev Wiki. `https://wiki.osdev.org/PE`. [Online; accessed 13-April-2020].

[3] Michael Sikorski and Andrew Honig. *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.

[4] Symantec. Internet security threat report volume 24, 2019.

[5] Verizon. 2019 data breach investigations report, 2019.

[6] Ekta Gandotra, Divya Bansal, and Sanjeev Sofat. Malware analysis and classification: A survey. *Journal of Information Security*, 2014, 2014.

[7] Igor Santos, Felix Brezo, Xabier Ugarte-Pedrero, and Pablo G Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[8] Bojan Kolosnjaji, Ghadir Eraisha, George Webster, Apostolis Zarras, and Claudia Eckert. Empowering convolutional networks for malware classification and analysis. In *2017 International Joint Conference on Neural Networks (IJCNN)*, pages 3838–3845. IEEE, 2017.

[9] Renjie Lu. Malware detection with lstm using opcode language. *arXiv preprint arXiv:1906.04593*, 2019.

[10] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[11] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620, 2009.

[12] Mehadi Hassen and Philip K Chan. Scalable function call graph-based malware classification. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 239–248, 2017.

[13] Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9(1):35–47, 2013.

[14] Anh Viet Phan, Minh Le Nguyen, Yen Lam Hoang Nguyen, and Lam Thu Bui. Dgcnn: A convolutional neural network over large-scale labeled graphs. *Neural Networks*, 108:533–543, 2018.

[15] Jiaqi Yan, Guanhua Yan, and Dong Jin. Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 52–63. IEEE, 2019.

[16] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

[17] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 2020.

[18] Štěpán Dvořák. Detekce škodlivých soubor pomocí metod statické analýzy. B.S. thesis, České vysoké učení technické v Praze. Vypočetní a informační centrum., 2018.

[19] Part Guide. Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2:5, 2011.

[20] Vector 35 Inc. Binary ninja. `https://binary.ninja`. [Online; accessed 16-April-2020].

[21] Vector 35 Inc. Binary ninja - architecture agnostic function detection in binaries. `https://binary.ninja/2017/11/06/architecture-agnostic-function-detection-in-binaries.html`. [Online; accessed 16-April-2020].

[22] Yararules project. `https://github.com/Yara-Rules/rules`. [Online; accessed 20-April-2020].

[23] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[24] Sachin Jain and Yogesh Kumar Meena. Byte level n–gram analysis for malware detection. In *International Conference on Information Processing*, pages 51–59. Springer, 2011.

[25] Yuta Nagano and Ryuya Uda. Static analysis with paragraph vector for malware detection. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, pages 1–7, 2017.

[26] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[27] Royi Ronen, Marian Radu, Corina Feuerstein, Elad Yom-Tov, and Mansour Ahmadi. Microsoft malware classification challenge. *arXiv preprint arXiv:1802.10135*, 2018.

[28] Guanhua Yan. Be sensitive to your errors: Chaining neyman-pearson criteria for automated malware classification. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 121–132, 2015.

[29] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[30] Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. In *Advances in neural information processing systems*, pages 3844–3852, 2016.

[31] Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: Higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609, 2019.

[32] Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pages 1024–1034, 2017.

[33] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[34] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[35] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[36] Federico Errica, Marco Podda, Davide Bacciu, and Alessio Micheli. A fair comparison of graph neural networks for graph classification. *arXiv preprint arXiv:1912.09893*, 2019.

[37] Hyrum S Anderson and Phil Roth. Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*, 2018.

[38] Cisco Threat Grid. https://www.cisco.com/c/en/us/products/security/threat-grid/index.html. [Online; accessed 18-May-2020].

[39] Reversing Labs. Titanium cloud file reputation service. https://www.reversinglabs.com/products/file-reputation-service.html. [Online; accessed 18-May-2020].

[40] Talos Intelligence blog. https://blog.talosintelligence.com/. [Online; accessed 23-March-2020].

[41] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[42] Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[43] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning*, pages 1188–1196, 2014.

[44] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnnexplainer: Generating explanations for graph neural networks. In *Advances in Neural Information Processing Systems*, pages 9240–9251, 2019.

Appendix B
Thesis attachments

Attachment to this thesis contains Python code, which was used for the training of the models. It is necessary to supply the code with your data. We can not, unfortunately, disclose our data due to privacy regulations. The training script saves results in AWS S3. The results contain predicted and true labels, the model, and metrics exported for Tensorboard. We attach tensorboard data for all the models we trained.

The files are structured as follows:

- `/dvorast6_diploma_thesis.pdf` is an electronic version of this thesis.
- `/src/` contains Python scripts used to train the models.
- `/tensorboard/` contains output generated into a format readable by Tensorboard.