BABEŞ-BOLYAI UNIVERSITY CLUJ-NAPOCA
FACULTY OF MATHEMATICS AND
COMPUTER SCIENCE

PHD THESIS SUMMARY

Machine learning methods in malware analysis

Attila Mester

Scientific supervisor: prof. dr. Anca Andreica

2025

Keywords: static malware analysis, reverse engineering, IDA, Radare2, APT attribution

Abstract

Cyber threats are a constantly growing concern for both organizations and individuals. According to recent statistics, this number is around three new potentially malicious files arising each second, globally. This amount of data is impossible to process manually by cyber security companies, therefore, automated tools are used to help in the identification and classification of these threats. A security tool's threat intelligence report contains tags such as verdict (malicious or not), detection labels, and also the *attribution*, regarding the file's family or the group of attackers that created it.

This thesis presents a comparison between two industry-leading disassembler tools, IDA and Radare2, focusing on the static call graph generation mechanism. Then, we present new feature extraction methods and algorithms with the aim of attributing a malware sample to a certain family. Two main directions are explored, malware clustering and classification, both leveraging the static call graph of the samples.

First, we propose a new signature extraction method from the static call graph, which is used to group the malicious samples in a graph based on common fingerprints. After clustering the files with community detection algorithms, we show that the resulting clusters are highly homogeneous with regard to the samples' families.

Second, two new classification methods are presented: one of these is based on graph convolutional neural networks, trained on static call graphs, while the other offers an encoding method to convert the static call graph into RGB images, thus, transforming the malware classification problem into an image classification one. These experiments also show insight into the relation of families and packers, demonstrating the effect of packers in the classification process.

We also publish a new malware dataset on Kaggle, together with various analysis information of two other state-of-the-art datasets in the field, MalImg and BODMAS.

Contents

Chapter 1

Introduction

Digitalization has increased the number of online threats we face each day. Although the majority of cyber attacks target larger corporations, businesses, and government organizations, where the potential benefit of compromising a system is significantly larger, individuals are also at risk. The common term to describe these threats is malware, which stands for malicious software, meaning a type of computer program or file which aims to do harmful actions on a computer. According to AV-TEST[1], the number of new threats appearing on a daily basis is approximately 3/sec, or $300k$/day.

1.1 Objectives

We aim to develop new methods for malware analysis, clustering and classification based on static call graphs of malicious files. Our focus is on fast, scalable processing of these entities, to enable the integration of our methods into real-world systems. We focus on methods that can grasp the structure of the graph in a scalable way, either by extracting locality-sensitive signatures from the graph, or by feeding the graph into neural networks that can learn useful patterns of its topology.

Furthermore, since the malware research domain lacks the abundance of publicly available datasets, we aim to enrich the community by publishing our datasets, the source code of our methods, and a comprehensive analysis of already existing datasets.

[1] `https://www.av-test.com`

1.2 Contributions

The contributions of this thesis are the followings (summarized on our GitHub Page[2]).

(i) Three novel methods are presented in the field of malware classification. We introduce a novel signature extraction method based on the static call graph of an executable. These signatures are then used to cluster the files by applying Louvain community detection algorithm on the malware graph. We demonstrate that our method can cluster malicious files into homogeneous groups according to their ground truth family label, by comparing them to industrially used patented signatures.Furthermore, we present two novel approaches for classifying malware code into families. The first method uses graph convolutional neural networks to learn the patterns of the static call graphs of malicious files. The second method transforms the static call graph into an image, and uses well-known CNN architectures to classify the malware samples.

(ii) We publish a GitHub project, *malflow*[3], that contains the source code of our static call graph generation algorithm. The project also contains the source code of our proposed image generation algorithms and the instruction encoding schemes.

(iii) We analyze existing state-of-the-art public malware datasets, considered benchmarks in the field, namely BODMAS and MalImg, and also introduce a new dataset, Internal Bitdefender Dataset (IBD), under the public GitHub project *malflow*. The information we publish contains the following:

- Radare2 disassembled objects, containing the full static call graph, with function names, calls, and also complete instruction information, such as the mnemonic, prefixes, operands, and the address of the instruction;
- various statistics regarding the instructions extracted from these malicious files, such as their distribution, and order of appearance according to the simulated execution of the file;
- packer information and entropy, obtained by scripted use of DIE tool;
- three types of RGB images for each sample, based on our different encoding schemes which assigns a pixel to an instruction. These images are used in

[2] `https://attilamester.github.io/call-graph`
[3] `https://github.com/attilamester/malflow`

our second classification approach, where we transform the static call graph into an image, and use state-of-the-art CNN architectures to classify the malware samples.

(iv) We reveal high correlation between APT families and the packer used to obfuscate the executable file. These observations are published for three different datasets spanning over nearly 15 years, MalImg, BODMAS and IBD, and are published in the context of the project *malflow*.

1.3 List of publications

The evaluation standards used are those valid as of October 1st, 2018. The list of conferences, as well as their categorization, is based on the international *CORE* conference rankings[4]. The journal list is the one used by *UEFISCDI*[5] for awarding articles published in international scientific journals.

- **A. Mester**, A. Pop, B. Mursa, H. Grebla, L. Diosan, C. Chira (2021). *Network Analysis Based on Important Node Selection and Community Detection*. Mathematics, 9.18, 2294
- **A. Mester**, Z. Bodó, P. Vinod, M. Conti (2025). *Towards a malware family classification model using static call graph instruction visualization*. 18th International Conference on Network and System Security, Network and System Security, 167–186, Springer Nature Singapore
- **A. Mester**, Z. Bodó (2021). *Validating static call graph-based malware signatures using community detection methods*. 29th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning, ESANN 2021, 429–434
- **A. Mester**, Z. Bodó (2022). *Malware classification based on graph convolutional neural networks and static call graph features*. Advances and Trends in Artificial Intelligence. Theory and Practices in Artificial Intelligence: 35th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, IEA/AIE 2022, 528–539, Springer
- **A. Mester** (2023). *Malware analysis and static call graph generation with Radare2*. In Studia Universitatis Babeș-Bolyai Informatica, 68.1, 5–20

[4] `https://portal.core.edu.au/conf-ranks`
[5] `https://uefiscdi.gov.ro/scientometrie-reviste`

Chapter 2

Malware analysis

The aim of this thesis is the static analysis of malware in the form of portable executable (PE) files. PE files can carry a vast amount of different attack techniques, hence they can also contribute enormously to the process of gathering threat intelligence about the origin of the attack. One such exceedingly valuable piece of information is called *attribution* in literature [Steffens, 2020].

Attributing APT information to cyberattacks is a high priority of threat intelligence organizations. The attribution covers information about the attack's authors and the deployed malware families. This information can be leveraged to identify the attackers, to reason their intentions, and to prioritize certain security measures. Attribution of attacks boils down to attribution of individual files and events. Manual analysts cannot process the vast amount of files and events that trigger alarms in real-time cybersecurity solutions. Therefore, automated analysis of malware samples is essential for antivirus products.

There are fundamentally two options regarding the analysis of an executable file: static and dynamic analysis. These methods assume the use of either a sandbox environment – which is often expensive and time-consuming, or a disassembler tool such as IDA, Radare2, Ghidra, etc., as detailed in Chapter 3. Both methods may be carried out manually by analysts, or automatically, using automated scripts and various tools.

Static analysis is limited to the inspection of the file content on the disk, without executing it. This method is faster and less resource-intensive than dynamic analysis, providing simple binary file properties (e.g. file size, section information, import and export table, flags) and also more complex features like byte distribution, disassem-

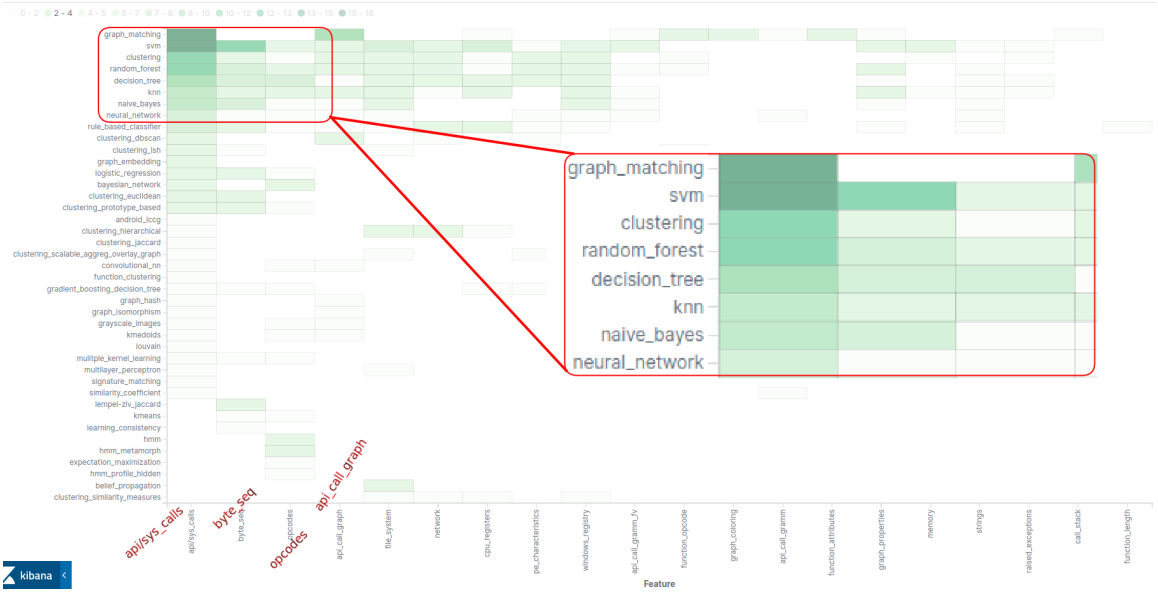


Figure 2.1: The most commonly used algorithms are applied on features selected from API/system calls. This plot aggregates the results of a survey paper [Ucci et al., 2019] categorizing approximately one hundred research papers in the field of malware analysis according to the extracted features and applied algorithms.

bled instructions, and their more complex representations such as control flow graph (CFG), or call graph. However, it may not be able to detect all the malicious behavior of the file, especially if the malware is encrypted or obfuscated.

Dynamic analysis necessitates a sandbox technology, which offers an isolated environment for the malware to be executed in, simulating a real-life scenario of infection. The sandbox will track the events triggered the this malware file, such as file system events, API calls, registry changes, network activity, disk operations. This type of analysis has the advantage of offering real events related to the activity of the malware, which will form its natural traits. However, its practicality may be considered limited due to the longer analysis time which depends on the malicious payload code, which may of may not be activated [Quertier et al., 2022].

The visualization in Figure 2.1 shows us the frequency of algorithm–feature pairs in the vast literature overview of [Ucci et al., 2019]. This shows us the possible new research directions, but most importantly the fact that the most popular methods involved applying some kind of graph matching algorithm, SVM method or any kind of clustering on data extracted from API/sys calls. This determined the direction of our research, aiming to explore the possibilities of leveraging the static call graph feature of an executable file.

Chapter 3

Disassembler tools

Analyzing malware often means the inspection of behavior of the PE file. Dynamic analysis can offer the inspection of the actual behavior of the file when executed, but it is time-consuming and might not be feasible for large datasets (see Chapter 2). When focusing on behavioral analysis, a static approach can be more efficient, as it can offer a quick insight into the file's behavior without the need to execute it, by analyzing the assembly code of the file. For this purpose, one can use various disassembler tools to reverse engineer the binary file and obtain its assembly code.

In this chapter, we describe two of the most popular disassembler tools, IDA and Radare2, and compare them in terms of their functionality and usability regarding the generation of static call graphs of executable files. These findings are published in our paper [Mester, 2023].

3.1 Overview of disassembler tools

IDA is an interactive disassembler[1], and one of the leading disassembler tools in the industry [Nar et al., 2019, Yin et al., 2018]. It has a rich GUI and offers a wide range of scripted functionalities as well, available through the IDA scripting language. Although its use is questionably versatile and offers a powerful analysis tool for reverse engineering, it has its limitations as well. One of the major drawbacks is that IDA is a commercial software, and the scripted functionality is available only in the paid version, IDA Pro.

There is a multitude of alternative disassembler tools: Binary Ninja

[1] `https://www.hex-rays.com/products/ida`

[Priyanga et al., 2022], Hopper [Andriesse et al., 2016], Relyze [Wenzl et al., 2019], x64dbg[2], ODA[3], etc. One of the most popular alternatives is Ghidra[4], available on Windows/Linux, developed by NSA's Research Directorate under Apache License (FOSS). According to various researches, it is a leading alternative to IDA Pro [Shaila et al., 2021, Koo et al., 2021]. The downside of this tool which made our choice of another alternative is the difficulty in using its scripted API call/graph generation.

Radare2[5] is also available on Windows/Linux (FOSS), and offers a lightweight alternative to Ghidra, while being able to integrate Ghidra decompiler *r2ghidra*[6]. It can be used from CLI and also GUI, offered by Cutter[7]. A major power of this tool is the Python binding *r2pipe*[8], which offers extensive APIs for static analysis, including call graph inspection. Radare2 (also referred to as *r2*) has also great popularity in the cyber tech domain [Massarelli et al., 2019, Cunningham et al., 2019, Gibert et al., 2020, Cohen, 2019, Kilgallon et al., 2017].

3.2 Generating the call graph with IDA Pro 6 and Radare2

IDA Pro 6

With its functionality of scripting in IDA Pro 6, we use two APIs to generate text files in GDL format. *GenCallGdl* generates the call graph structure (i.e. a node represents a function, an edge marks a function call), while *GenFuncGdl* generates the execution flow chart (i.e. a node contains an instruction list and the directed edges represent a jump of execution between these blocks). The final call graph is a processed version of both of these files, as detailed in our previous works [Mester, 2020, Mester, 2023].

[2] `https://x64dbg.com`
[3] `https://github.com/syscall7/oda`
[4] `https://ghidra-sre.org`
[5] `https://www.radare.org`
[6] `https://github.com/radareorg/r2ghidra`
[7] `https://cutter.re`
[8] `https://r2wiki.readthedocs.io`

Radare2

Radare2[5] offers a clear description of its installation on their Github page[9] and has plenty of documentation and community support on their Wiki page[10] and their official e-book [Radare, 2009]. After installation, Radare2 can be invoked using the *radare2* or *r2* commands, specifying a path to a PE file. In this CLI, we are offered a variety of commands and tools for analyzing sections, imports, exports, entry point information, blocks, function calls, for seeking certain parts of the binary, and much more – also, each command has a helper interface invocable by appending "?" after the respective command. Radare2 works with the concept of flags, i.e. a bookmark at an offset like "fcn." or "sym.imp", meaning that every offset considered interesting by Radare2 will be assigned a corresponding flag to it, e.g. strings, functions, imports, and much more. Analysis of a binary PE file can be started by the command "aaa", which analyzes all the flags in the file. Since in this work we were focused on the analysis of the static call graph, we will detail commands which are related to the analysis of the call sequences, function blocks, and entry points.

The majority of these *r2* commands have multiple output formats, available by specifying a formatter at the end of the command – such as the default ASCII art, or "j" for *json*, "d" for *dot*, "b" for "Braile art" i.e. short overview/bird's eye plot, or "w" for an interactive plot – highly useful for debugging purposes, similar to a *matplotlib* plot.

When generating the static call graph of a PE binary using Radare2, we leverage multiple *r2* commands to obtain the final graph object. Radare2 offers Python bindings via the *r2pipe* package, which simply enables the pipeline of multiple *r2* commands without the need to open and load the file each and every time. We start the analysis by calling "aaa" command. Then, we collect entry point nodes, i.e. function blocks by calling "ie". The *r2* commands that we use for call graph analysis are part of the "ag" command group.

The structure of the call graph is provided by the "agC" command, where we also specify the desired DOT format using the "d" flag. The full reference graph (e.g. imports) is offered by "agR" command. It is important to mention that none of these two commands include node-level information regarding to the instruction list. In order to obtain the assembly code of each subroutine, we call "agf" command on

[9] `https://github.com/radareorg/radare2`

[10] `https://r2wiki.readthedocs.io`

each node of the call graph. In a similar manner to generating the call graph using IDA Pro 6, merging the output of "GenCallGdl" and "GenFuncGdl" [Mester, 2020, Mester and Bodó, 2021, Mester and Bodó, 2022], the same logic applies in Radare2 as well. We need to gather information from both the global function graph ("agC") and global reference graph ("agR"), and also need to analyze separately each function block ("agf") in order to obtain the final, complete call graph.

3.3 Comparing the call graphs obtained by IDA Pro 6 and Radare2

One key difference between IDA Pro 6 and Radare2 is that in the former, we needed to call only 2 APIs, while in the latter, we need to call a multitude of *r2* commands – $O(n)$ where n is the number of function blocks. The unexpected revelation is that despite all these aspects mentioned, Radare2 scans the binaries much faster than IDA, and the call graphs generated by these two tools are almost identical [Mester, 2023].

3.4 Conclusions

This chapter presented the comparison of two disassembler tools, IDA Pro and Radare2, in terms of their static call graph generation capabilities. While IDA offers two API calls via its scripted language to simply generate the call graph, Radare2 requires a more complex approach, using the *r2pipe* library, i.e. its Python binding. In both of these cases, several steps need to be taken in order to construct a final, global call graph of the sample – naturally, this may be changed according to specific requirements and personal needs. During the experiments, a public dataset was used in order to offer full transparency of the results. The call graphs were compared from various perspectives, both topological aspects, i.e. the function calls, and also node-level criteria, i.e. the instruction list of each subroutine. The results claim that there is no significant change in the output of IDA and Radare2 disassemblers, however, the latter offers a faster, more stable way of scripted analysis which is suitable for a production environment where performance is a key aspect. These results were published in Studia Universitatis Babeș-Bolyai Informatica journal [Mester, 2023].

Chapter 4

Application of community detection in malware analysis

This chapter presents the application of community detection as a method to cluster malicious files according to their similarity [Mester and Bodó, 2021].

A community means a set of nodes, with the following properties: (i) they form a connected subgraph; (ii) nodes in community A have common properties (topical and topological features); (iii) nodes in community A have more common properties with other nodes in A than with those found in community B.

A detailed analysis of various concepts and metrics applicable in network science and community detection can be found in our previous work [Mester et al., 2021]. In this work, we conclude that among various existing hierarchical clustering algorithms, modularity optimizations, etc., the Louvain algorithm [Blondel et al., 2008] is a preferable option for large networks, due to its $\mathcal{O}(\ell)$ complexity.

In this chapter, we present a method where the aim is to build a network of malware samples, each node representing a virus, and links are drawn if there are common fingerprints – the fingerprint selection method is the crucial part, our core contribution to the domain. We propose a method to extract signatures from the executable binary of a malware, in order to query the local neighborhood in real-time. The signatures are obtained via static code analysis, using function call n-grams and applying locality-sensitive hashing techniques to enable the match between functions with highly similar instruction lists.

4.1 Related work

Based on a recent survey [Ucci et al., 2019], one can conclude that the most frequently used features are API/system calls, byte sequences and API call graphs. Using the API graph features, it is common to apply graph matching algorithms or calculate graph edit distances (GED) [Park et al., 2010]. Another approach is to build a feature vector of the graph based on n-grams [Dahl et al., 2013].

Convolutional neural networks are also successfully applied to detect patterns in (static) opcode and (dynamic) system call sequences for identifying malicious code [McLaughlin et al., 2017, Martinelli et al., 2017]. Malware clustering based on locality-sensitive hashing was already experimented in [Oprisa et al., 2014]. In [Hassen and Chan, 2017], LSH is applied on local subroutines, using minhash signatures to approximate similarity between the instruction sets of two subprograms.

Our method extracts multiple fingerprints from the call graph, enabling the continuous processing and clustering of new incoming samples, without the need to retrain the underlying model.Another key aspect of our approach is the validation of the industrially used features of [Topan et al., 2013] as well.

4.2 N-grams for malware clustering

The fingerprints are extracted from the static call graph, obtained by IDA Pro 6 disassembler[1] – see Section 3.2. A fingerprint is an n-gram in the resulting call graph: unigram meaning a codeword for a subroutine, while bigram standing for a caller–callee codeword pair.

In order to measure the usefulness of the generated signatures, community detection methods are applied on the common fingerprint-based malware graph – the nodes represent the malware and an edge indicates common fingerprints – as explained in Figure 4.1. One such malware graph can be seen in Figure 4.2, where the nodes are colored according to the Louvain communities obtained by Gephi[2].

We consider the following requirements: (i) the malware graph should consist of dense components (i.e. communities), with as few edges between them as possible – this means the separation of the malware groups from each other; (ii) ideally, the nodes of a component should share the same family label.

[1] `https://www.hex-rays.com/products/ida`
[2] `https://gephi.org`

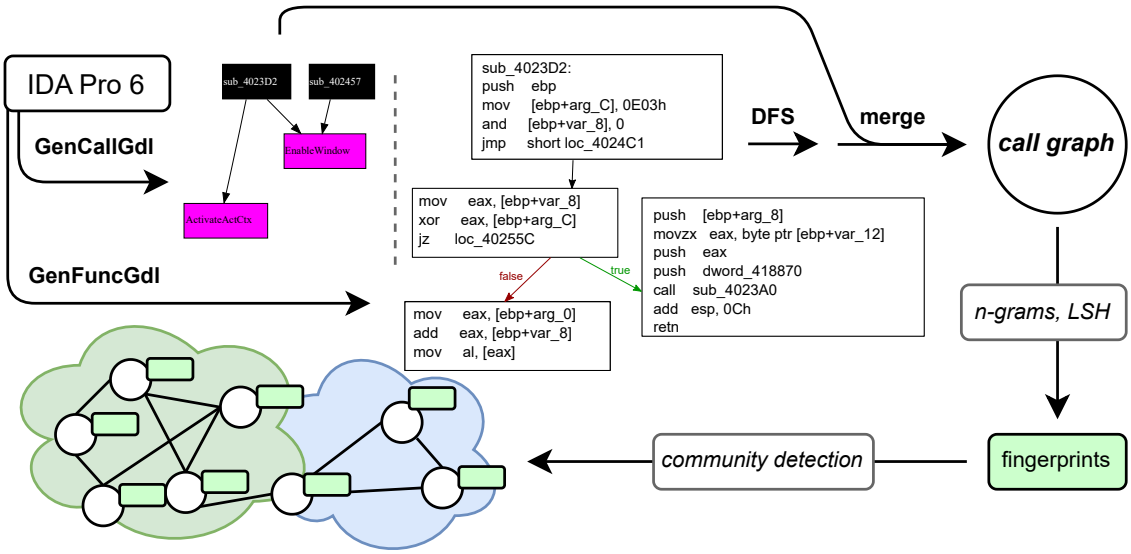


Figure 4.1: Pipeline of generating the fingerprints and validating these by applying community detection methods on the malware graph.

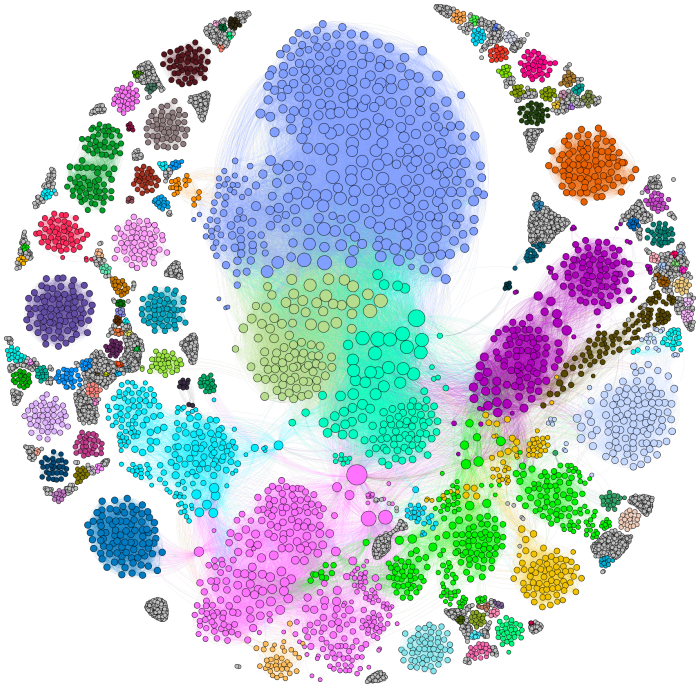


Figure 4.2: Louvain communities of the malware graph according to common fingerprints (internal Bitdefender dataset of 7977 samples from 254 families, having high family label confidence). Layout: Gephi's Force Atlas 2. The method proved to be well suited for family clustering, measured by the partition homogeneity score.

	Homogeneity	Completeness	Majority label
(a)	0.833	0.765	85%
(b)	0.632	0.704	89%

Table 4.1: Family-based numeric evaluation of the malware graphs.

4.3 Evaluation methodology

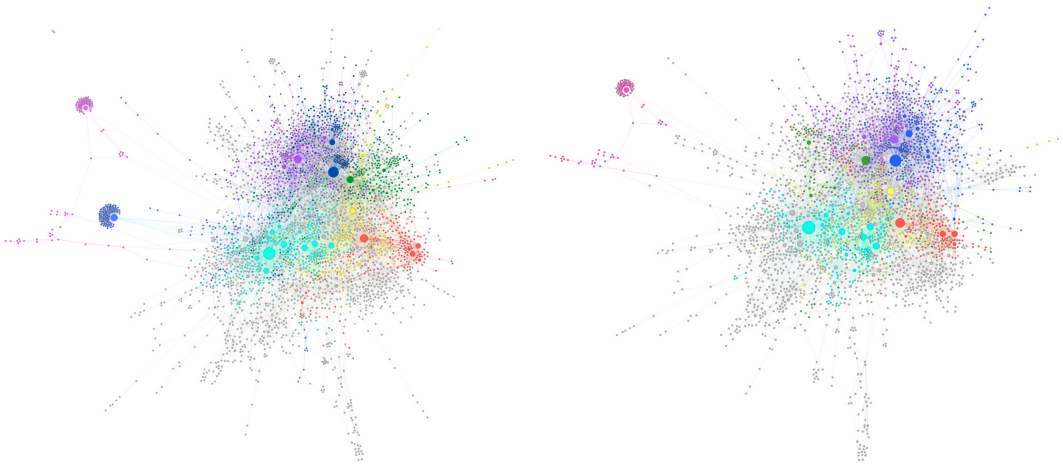
The most successful and frequently used community detection algorithms are Clauset–Newman–Moore (CNM), label propagation algorithm (LPA), Louvain and Infomap [Fortunato, 2010] – therefore, we chose these for validating our approach. In order to evaluate the topological properties of the obtained malware graph, we calculate *modularity*, *coverage* and *performance* scores [Fortunato, 2010]. Since the family distribution of the clusters is similarly important, we also measure the majority percentage of the labels within each cluster.

4.4 Hyperparameters

In the experiments, we used Python 3.6 (*networkX* library), IDA Pro 6, Graphviz, Gephi 0.9.2. We carried out experiments considering the following hyperparameter combinations: subroutine instruction n-grams (1–3-grams, 2–3-grams or trigrams); number of random hyperplanes (8 or 16); projection partition (by projecting a vector onto a hyperplane and taking the sign of the projection to generate the hash codewords, the distance information is lost – enabling this parameter sets distance intervals of $[0, 10]$, $(10, 100]$, $(100, 1000]$ and > 1000, and symmetrically with negative signs, to use a finer partition of the space); call graph n-grams (unigrams or bigrams). On the final graph, the following filters are applied: frequency of fingerprints ($[2, 80]$, $[2, 100]$, $[3, 100]$, $[10, 100]$ or $[10, 400]$); edge weight (minimum 5, 10, 100, 200 or 1000). These parametrizations resulted in a total of 260 runs.

4.5 Dataset

The private Bitdefender dataset consists of 7977 samples from 254 families, a family having 30.3 ± 41.96 samples on average. The number of subroutines per sample is on average 1163 (median of 354 and maximum of 65 060) – summing up to a total



Sample s_1 Sample s_2

Figure 4.3: Comparison of two call graphs from the same family within the same community (the extra function group on the left graph shows a *winsock2* package).

number of 9 284 242 subroutines with 650 225 unique mnemonic sequences.

4.6 Conclusions

We calculated the above-mentioned evaluation metrics for different malware graphs, based on the following fingerprints:

(a) industrially used fingerprints [Topan et al., 2013] (4745 nodes, 55.9k links)

(b) fingerprints described in this work: instruction 2–3-grams, 8 hyperplanes, projection partition, call graph bigrams, $[2, 100]$ frequency filtering and min. weight of 100 (4502 nodes, 114k links).

Results metrics are compared in Table 4.1. Based on the experiments, the following conclusions can be drawn: *(i)* the industrially used fingerprints can cluster families with high modularity and majority label percentage scores; *(ii)* configuration **(b)** provides similarly good results as **(a)**; *(iii)* instruction n-grams, sequences yield better representation than simple unigrams; not only the code of the subroutines, but also the sequentiality of these subroutines can characterize malware families.

To summarize, we have presented a novel method to extract fingerprints from static analysis of a malicious sample, by applying LSH on its subroutines.We have shown that our method can be used to cluster malware families with high modularity and majority label percentage scores [Mester and Bodó, 2021].

Chapter 5

Malware classification: attribution

This chapter presents the problem of malware classification, i.e. family attribution [Steffens, 2020]. We present two novel methods, one based on graph convolutional networks and the other on image classification, both of these leveraging the static call graph of the malicious file.

5.1 Introduction

One of the most relevant and valuable label a malicious sample can be assigned is the family and author information [Ucci et al., 2019, Tahir, 2018, Steffens, 2020].

Our first method leverages the topology and also node-level information of the static call graph by training a GCN model to classify families [Mester and Bodó, 2022]. This model was trained on a real dataset consisting of thousands of malicious samples from 223 families – a number significantly larger than the families used in the literature.

The other method converts the malicious sample into an image, and executes the family classification based on an image classification task. The novelty of this method lies in the way the images are created, based on a specific traversal of the instructions from the static call graph, which simulates the theoretical execution flow of the program [Mester et al., 2025].

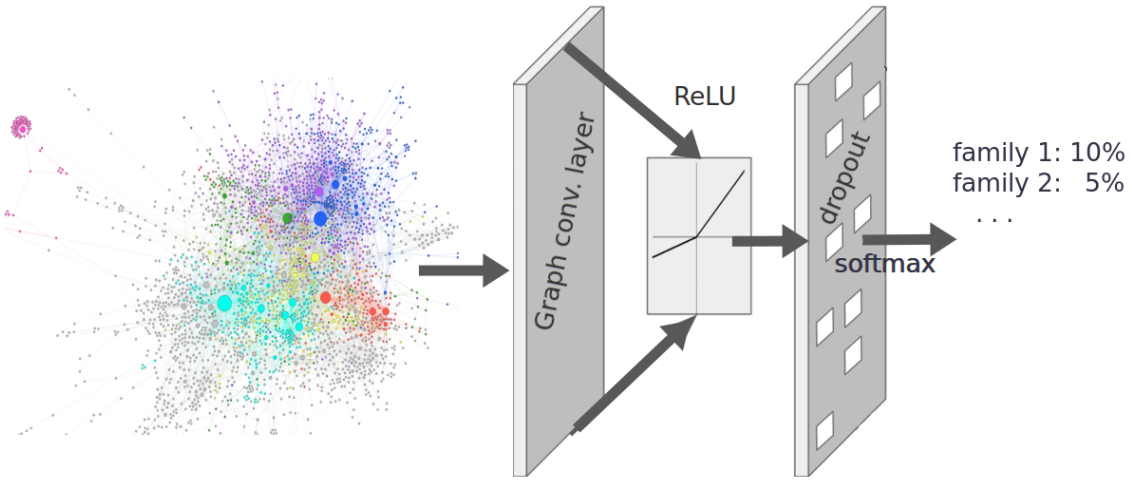


Figure 5.1: Project outline for the attribution of malware samples using GCN on the static call graph [Mester and Bodó, 2022].

5.2 Attribution using GCN on the static call graph

5.2.1 Related work

API calls (dynamic as well as static) are by far one of the most extensively used features in the literature for malware classification [Ucci et al., 2019]. Graph convolutional neural networks (GCN), deep GCN models (DGCNN) show an increasing popularity in cyber threat intelligence, likely due to the novelty as well as the benefits of these approaches. Furthermore, we can also conclude that malware detection methods are in general validated using only around 10 classes. API calls and functions obtained from static analysis are used for malware detection and family classification in [Dam and Touili, 2017, Hong et al., 2018, Phan et al., 2018, Hong et al., 2019]. Similarly, these features are used in node and graph embedding techniques [Jiang et al., 2018, Hong et al., 2019, Yan et al., 2019] and GCN methods [Li et al., 2021]. It is important to mention the size of the datasets and the number of classes used in the classification process. Simple binary classification (malicious/benign) is applied in [Dam and Touili, 2017, Jiang et al., 2018, Phan et al., 2018, de Oliveira and Sassi, 2021]. [Hong et al., 2018] mentions 7 classes of authors, [Hong et al., 2019] classifies samples into 6 families, while [Tang and Qian, 2019] into 9 families, and [Yan et al., 2019] mentions 12 families.

5.2.2 Theory of graph convolutional networks

Convolutional neural networks (CNNs) are shift-invariant neural networks extracting local features from signals using the convolution operation. Graph convolutional networks (GCN) generalize CNNs to graph structures using convolutions from spectral graph theory [Kipf and Welling, 2016, Wu et al., 2019, Bruna et al., 2014, Henaff et al., 2015, Chung, 1997].

In this research we use the GCN model introduced in [Kipf and Welling, 2016] based on Laplacian smoothing, i.e. averaging every point over its neighbors in the graph. The propagation rule of this spatial GCN is the following,

$$\mathbf{H}^{(i+1)} = \sigma\left(\tilde{\mathbf{A}}\mathbf{H}^{(i)}\mathbf{W}^{(i)}\right) \tag{5.1}$$

where $\mathbf{H}$ denotes the (embedded) data representation, which initially contains the input features, i.e. $\mathbf{H}^{(0)} = \mathbf{X}$, $\tilde{\mathbf{A}}$ is the symmetrically normalized adjacency matrix, $\mathbf{D}^{-1/2}\mathbf{A}\mathbf{D}^{-1/2}$, containing self-loops, and $\mathbf{D} = \mathrm{diag}(\mathbf{A} \cdot \mathbf{1})$ is the diagonal degree matrix. The matrix $\mathbf{W}$ denotes the weights of the neural network, and σ is a non-linear activation function, usually ReLU [Goodfellow et al., 2016]. The above GCN model produces an embedding vector for each node, therefore these have to be summarized for the entire graph, for which a common choice is to use average pooling.

Graph learning contains the following three types of tasks [Zhou et al., 2020]: (i) *node-level* tasks, e.g. node classification, (ii) *edge-level* tasks such as link prediction, and (iii) *graph-level* tasks, like graph classification.

5.2.3 Family classification using GCN

The call graph is in fact the combination of the sample's control flow graph and function call graph – details in [Mester and Bodó, 2021, Mester, 2020]. Node-level features are the LSH codewords used in [Mester and Bodó, 2021] – each node will have a 8-dimensional vector, representing its 600-thousand-long instruction distribution vector's projection onto 8 random hyperplanes [Charikar, 2002].

We built a GCN using the PyTorch library, specifically, the geometric package[1]. The model resulting the best F_1-scores while keeping low variance is as follows (using the official *torch* layer names): three *GCNConv* layers, each followed by *ReLU*, then *Dropout*, then a *GCNConv* followed by *Dropout* and a *global_mean_pool* layer, finally,

[1] `https://www.pyg.org`

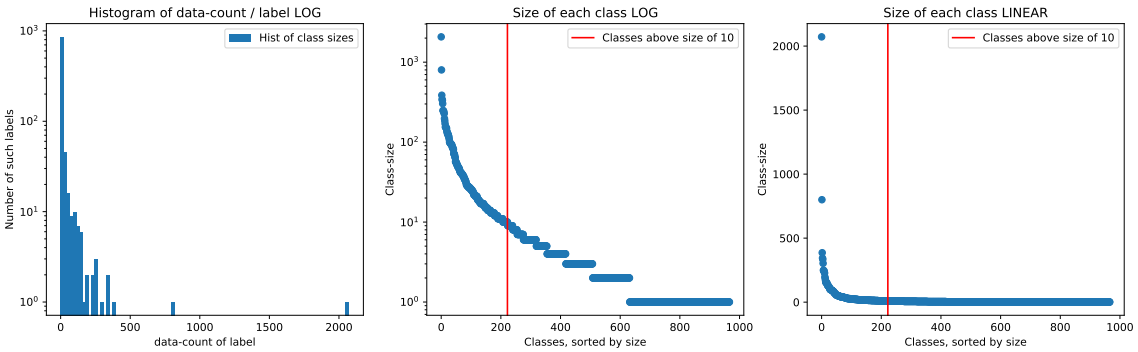


Figure 5.2: Distribution of family sizes within the dataset of $15k$ samples.

a fully connected layer, serving the output probabilities. For model optimization, we use the *CrossEntropyLoss* loss, and the *Adam* optimizer, with a learning rate of 0.01.

5.2.4 Comparison: node-level features vs. topological features

Experiments were conducted on the GCN model with and without node-level features, multilayer perceptron (MLP) model on the node-level features, and also on other features described in [Yan et al., 2019].

By training the same GCN model without node-level features (i.e. based only on the call graphs' adjacency matrix), we may answer the question whether the LSH codewords are truly useful when classifying families.

Eliminating the GCN model, and training instead an MLP on the node-level features, we could examine whether the LSH codewords can classify malware families with the same success (i.e. F_1-score) as the full GCN model – in other words, whether the topological features bring valuable information for the model to learn on. Similarly to the previously mentioned GCN model, for this MLP we used *CrossEntropyLoss* loss function, and *Adam* optimizer with a learning rate of 0.01.

Furthermore, we trained the same GCN/MLP models as before, but using other set of node-level features, the ones suggested in [Yan et al., 2019]. In this paper, the authors train a GCN model on the CFG of the samples, using as node-level features a 11-long vector, representing the instruction distribution (e.g. transfer, call, arithmetic, etc.). In our experiments we simulate this method by assigning each function a vector containing 14 numbers, according to the mnemonic distribution[2].

[2]x86 Assembly Language Reference: data transfer, *mov*, control transfer, *call*, arithmetic, compare, flag, bit and byte, shift and rotate, logical, string, I/O instructions; and in- and out-degree.

Model	Micro-F_1	Macro-F_1
GCN model with LSH codes	0.381	0.189
GCN model with features of [Yan et al., 2019]	0.614	0.392
GCN model without node-level features	0.204	0.003
MLP model with LSH codes	0.313	0.050
MLP model with features of [Yan et al., 2019]	0.242	0.020

Table 5.1: F_1-scores of each model on the test dataset.

5.2.5 Dataset

The private Bitdefender dataset contains 15 375 samples from 967 families. Filtering out the families with less than 10 samples, we obtained a dataset of 8620 samples from 223 families. This selection is illustrated in Figure 5.2.

5.2.6 Results

We used Python3, IDA Pro 6, GraphViz, PyTorch 1.10.0, Pytorch Geometric (pyg) 2.0.2, Tensorboard, on a system with Intel Xeon E5-2697A v4, 64 GB RAM, and a GeForce RTX 2080 Ti video card. Different combinations of hyperparameters were tested: $1-4$ hidden GCN layers with sizes of 64, 128 or 256, dropout probability: 0.2, 0.4 or 0.5. The best GCN model is as follows: 4 GCN layers of size 128, dropout of 0.5. From Table 5.1 we can conclude that GCN models with node-level features have the best F_1-scores, meaning that using both topological features of the static call graph and the feature vectors of the local functions yields the best results.

5.3 Attribution using CNN on the encoded static call graph

For accurate malware family classification, we propose mapping families to their common behavioral features based on patterns in their instruction images. We traverse the call graph's functions to compile a list of instructions, encoding each instruction into a pixel to generate the final RGB image. The pipeline of our work can be seen in Figure 5.3.

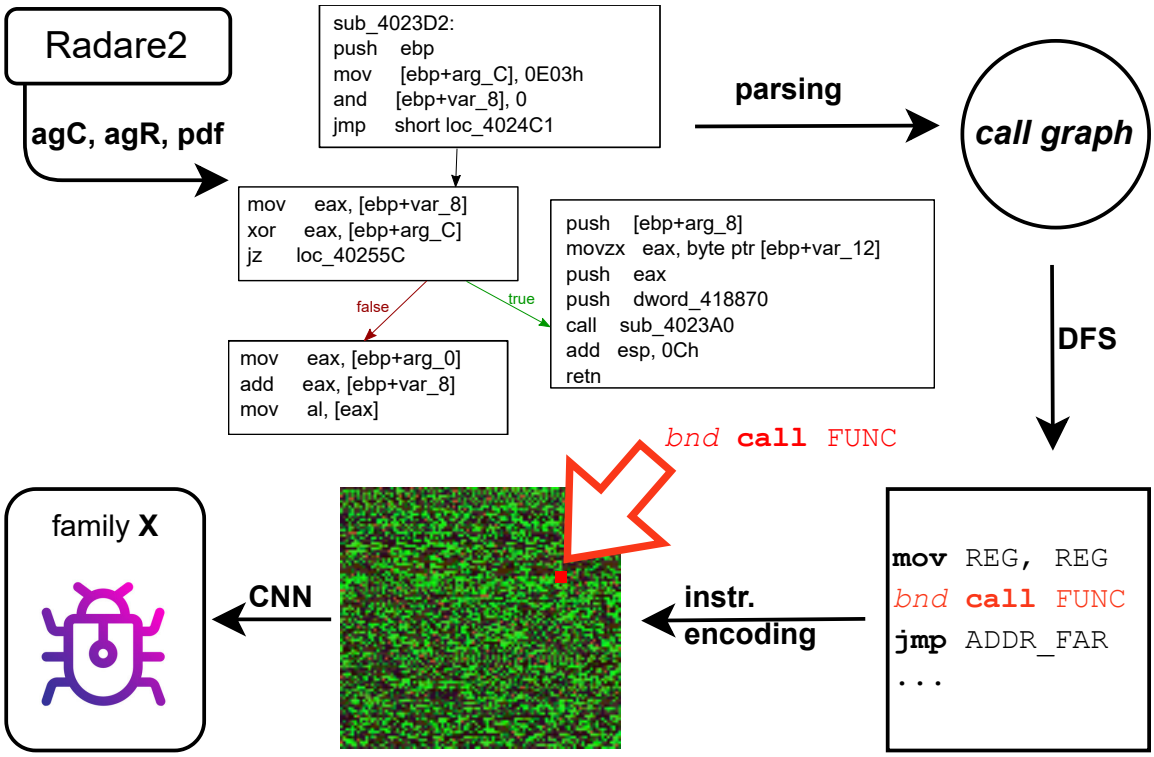


Figure 5.3: Our proposed method processes Radare2 output into a call graph, extracts instructions via DFS, encodes them as an RGB image, and uses it to train CNNs.

5.3.1 Related work

The first model using image-like representation was described in [Nataraj et al., 2011], the paper introducing also the widely used MalImg dataset. Here a grayscale 2D image is constructed from the binary file, each pixel representing one byte of the sample's binary. The works [Cui et al., 2018, Kalash et al., 2018] also carry out experiments using grayscale image representations of malware. Various other research papers build on the idea of representing the malware code by an image [Fu et al., 2018, Xiao et al., 2021, Yuan et al., 2020, Deng et al., 2023, Ni et al., 2018]. The images are usually fed to Convolutional Neural Networks (CNNs) as inputs, the most popular architectures being AlexNet [Krizhevsky et al., 2012], VGG [Simonyan and Zisserman, 2015] and ResNet [He et al., 2016].

5.3.2 Creating the static call graph

The static call graph generation necessitates a disassembler tool. We chose Radare2 (5.8.8 release) because of reasons detailed in [Mester, 2023]. Hereby we list the

key steps of generating the static call graph of a PE file using Radare2 (details on GitHub[3]). By using the `r2pipe`[4] Python package, we apply the `agCd` and `agRd` commands to yield the global call and reference graph. After traversing these graphs, we call for each function the `pdfj` command to obtain its instruction list. An instruction holds information about its bnd flag [Guide, 2011] and prefix (if any), its mnemonic and a list of parameters. We considered 11 prefixes (according to the Radare2 5.8.8 release), and 7 parameter types. The structure of an instruction is as follows:

$$\textbf{[bnd?] [prefix?] mnemonic [param1 [param2, ...]]}. \tag{5.2}$$

5.3.3 Converting the call graph into image

We intended to grasp the static behavioral features and patterns in the call graph of an executable, by creating an image based on the instructions in the call graph. This way, the image does not reflect all bytes from each section, only the relevant instructions, thus, eliminating the *noise*, compared to grayscale images based on hex dumps. We apply a depth-first search traversal (DFS) of the call graph to obtain the list of nodes, i.e. functions – this list reflects the execution order of these functions. We apply another DFS, now on the instruction level: in the order of these nodes e.g. *A*-*B*-*C*, we gather the instructions from A, but once we meet a *CALL*-like instruction, we jump to that address.After the recursive context finishes, we continue parsing the instructions in block A. Then, we move on to block B – if it was not visited already.

To encode one instruction into a pixel (i.e. a three-byte value, according to RGB images), we examined three distinct encoding schemes (details in [Mester et al., 2025]): (i) (FE) Full Encoding: mnemonic, prefix, bnd, two parameters; (ii) (PE1) Partial Encoding: mnemonic, prefix, bnd; (iii) (PE2) Partial Encoding: only mnemonic.

5.3.4 Training CNN models

We perform experiments on ResNet18, ResNet50 [He et al., 2016], ResNet1D [Hong et al., 2020], MobileNetV3 [Howard et al., 2019], GoogleNet [Szegedy et al., 2015], EfficientNet [Tan and Le, 2021], and DenseNet [Huang et al., 2017]. Experiments were carried out on four different types of

[3] `https://github.com/attilamester/malflow`
[4] `https://pypi.org/project/r2pipe`

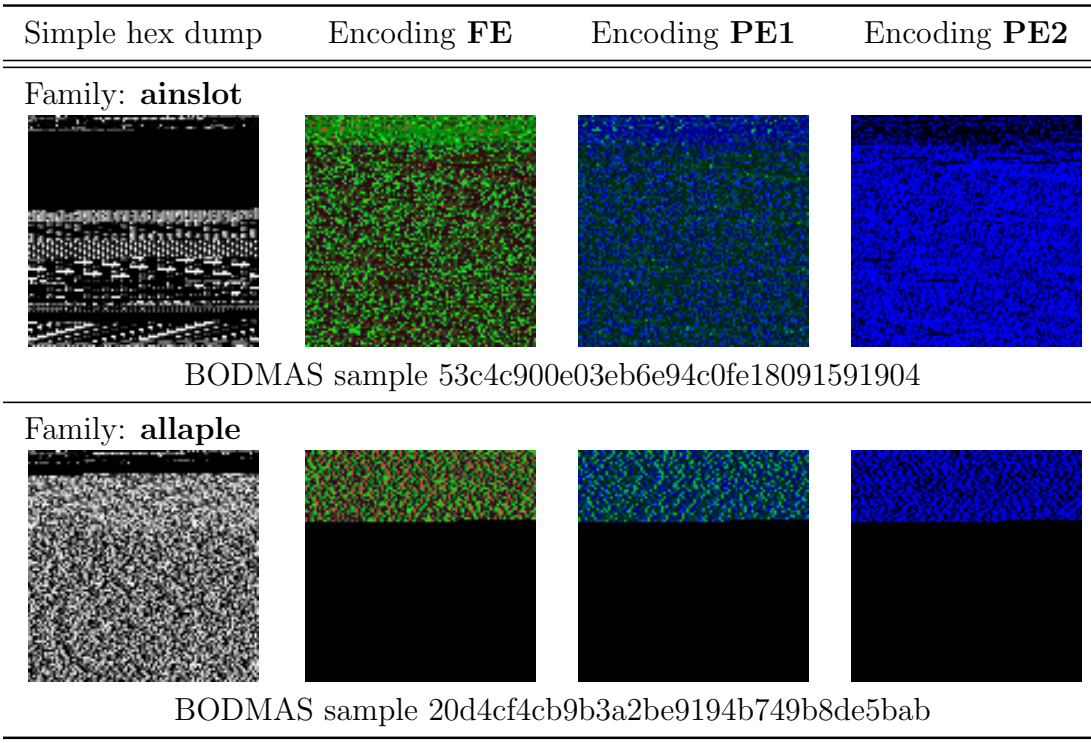


Figure 5.4: Comparing different instruction encodings (Section 5.3.3) and the simple hex dump image on different malware families.

images: hex dump-based grayscale images (similarly to [Nataraj et al., 2011]), and the three different types of instruction encoding schemes presented in Section 5.3.3. In Table 5.3 we show results according to **FE** scheme, as it performed best across all models. We applied a random search on the hyperparameter space: (i) *CNN architecture*; (ii) *Pre-trained (bool)* (on ImageNet); (iii) *Min. samples per class (int)* – set to 100 in Table 5.3; (iv) *Batch size (int)*. More than 200 models were trained, with runtimes ranging from 30 minutes up to 26 hours for the larger images.

5.3.5 Public datasets: MalImg, BIG, EMBER, BODMAS

Table 5.2 summarizes the properties of the publicly available malware datasets offering family labels. MalImg [Nataraj et al., 2011] contains $9.5k$ malicious samples' grayscale image based on their hex dump – categorized into 25 families [Zhan et al., 2023, Hai et al., 2023, Kim et al., 2023, Moussas and Andreatos, 2021, Gibert et al., 2019]. The EMBER dataset (detailed in Section 5.3.8) was first released in 2017 [Anderson and Roth, 2018], and later updated in 2018. It is a significantly larger database than the former, having $800k$ malicious samples' SHA256 and fea-

Dataset	Published	Binaries	Families	Samples(mal.)	EMBER	Disasm.	Image
MalImg	2011	❍	25	9458	❍ ✪	❍ ✪	● ✪
MS BIG	2015	❍	9	10 868	❍	●	❍
EMBER	2018	❍	◗	800 000	●	❍	❍
UCSB-packed	2020	●	❍	232 415	❍	❍	❍
SOREL-20m	2020	●	❍	9 962 820	●	❍	❍
BODMAS	2021	●	581	57 293	●	❍ ✪	❍ ✪
IBD	**2024**	❍	**47**	**18 756**	● ✪	● ✪	● ✪

Table 5.2: Public malware datasets: MalImg [Nataraj et al., 2011], MS BIG [Ronen et al., 2018], EMBER [Anderson and Roth, 2018], UCSB-packed [Aghakhani et al., 2020], SOREL-20m [Harang and Rudd, 2020], BODMAS [Yang et al., 2021] and IBD – our contribution, whithin *malflow*. ❍ = "not available", ● = "available", ◗ = "partially available", ✪ = "published by this work".

Model	Batch	Img.	Acc.	F_1 micro	F_1 macro
BODMAS, 57 **families**					
ResNet18	20	100×100	0.887	0.887	0.859
BODMAS, 57 **families, hex dump images**					
ResNet18	20	100×100	0.881	0.881	0.873
MalImg, 23 **families**					
ResNet18	20	100×100	0.722	0.744	0.737
IBD, 47 **families**					
ResNet18	20	100×100	0.872	0.872	0.784

Table 5.3: Performance of various models on BODMAS, MalImg and IBD.

ture vector [Maillet and Marais, 2023, Jia et al., 2023, Dener and Gulburun, 2023, Sandor et al., 2023, Quertier et al., 2022, Gao et al., 2022, Hussain et al., 2022, Yan et al., 2022, Yang et al., 2021, Feng et al., 2014]. BODMAS [Yang et al., 2021] represents the core of our experiments, with 57 923 malware binaries, each having a family label and the EMBER feature vector.

5.3.6 Dataset

The CNN models were trained on three different datasets: BODMAS, MalImg, and an internal Bitdefender dataset (IBD[5]). Figure 5.5 shows the distribution of families in the three datasets, demonstrating their high imbalance – we also tried data augmentation by invoking a metamorphic engine[6] to generate variations for small

[5] `https://kaggle.com/datasets/amester/malflow`

[6] `https://hub.docker.com/repository/docker/attilamester/pymetangine`

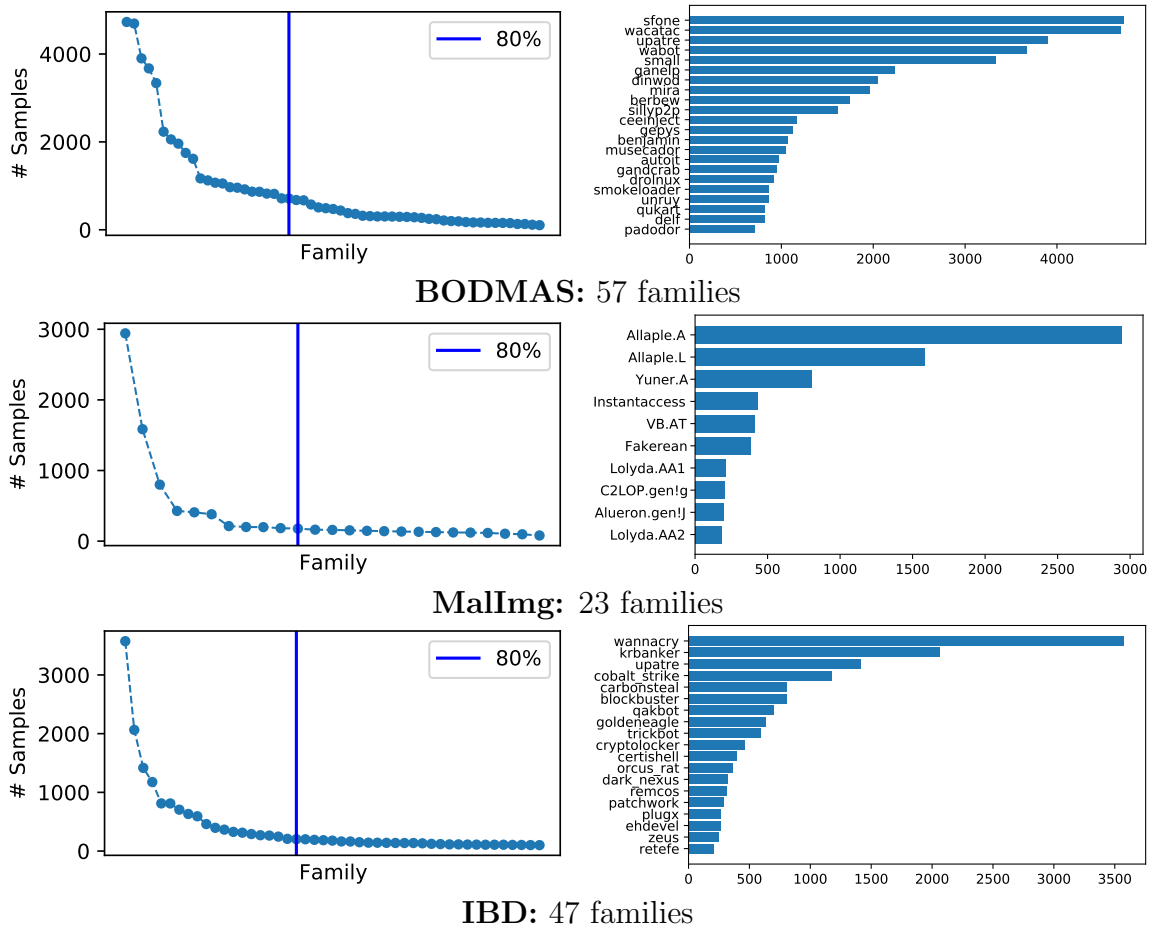


Figure 5.5: Family distribution in the datasets, and top families making up 80% of the data, used for training CNNs on instruction images.

families, yielding no significant improvement.

5.3.7 Results

We used Python3.8 with Radare2 5.8.8, and a GPU server with two RTX 2080 Ti graphics cards. Table 5.3 shows the result of the best models, with ResNet18 achieving the highest F_1 score. One can observe that larger images perform better than the smaller 30×30 ones, due to the amount of information they contain. We can formulate, that the best F_1 score achieved by these models is 0.887, performed by a ResNet18 on 100×100 images, using a batch size of 20 images, on the BODMAS dataset, and similarly on the IBD dataset as well. On the MalImg dataset, the winner was still a ResNet18 model, now with 0.744 F_1 score. Similar results have been obtained on the BODMAS with hex dump images – concluding that the files' structure,

data section and such other "noise" is still useful for family classification. Similarly, the information encapsulated in this "noise" coupled with its correlation with actual family labels (Chapter 6) is probably the reason why hex dump image-based classification on MalImg yields 0.98 accuracy [Nataraj et al., 2011] compared to the much lower score of only 0.72 with the instruction-based images.

5.3.8 Comparison with the EMBER baseline

The EMBER [Anderson and Roth, 2018] contains the aggregation of virtually all the static features from a PE file, such as header, section, import and export information[7].

Since our experiments first targeted the BODMAS [Yang et al., 2021] which also contains the EMBER features, a good comparison metric of our call graph instruction-based image classification model was to transform the images into vectors, by measuring the instruction mnemonic distribution, regarding 2019 mnemonics.

Based on various linear models (decision tree, random forest and SVM classifier), trained both on full datasets and filtered ones (containing only non-packed files), we conclude that packed files distort the classification performance in the case of mnemonic histogram model, but not in the case of EMBER features, which contains other binary file-based features as well. Hence, we formulated our hypothesis that packers may be correlated with malware families – confirmed in Chapter 6.

5.4 Conclusions

This chapter presented the role of supervised machine learning in the context of malware family classification. We presented a GCN-based approach to classify malware, using static features based on the call graph of a PE [Mester, 2020, Mester and Bodó, 2021]. The other method presented is about transforming the static call graph of a PE into an image and utilizing a CNN for classification [Mester et al., 2025] – presented at NSS 2024[8]. Another contribution is the IBD dataset[9], published on Kaggle[10] – containing Radare2 disassembled call graphs of the samples from IBD, MalImg and BODMAS, as well as other information such as packer, family, instruction distribution, and the encoded RGB images.

[7] `https://github.com/elastic/ember`
[8] `https://nsclab.org/nss-socialsec2024/papers.html`
[9] `https://github.com/attilamester/malflow`
[10] `https://www.kaggle.com/datasets/amester/malflow`

Chapter 6

Packers and malware families

This chapter explores the impact of using packers on executable files, in the context of static malware analysis. We discover an interesting correlation between packer tools and malware families within the analyzed datasets.

Packing executable files has no malicious intent per-se – originally, it was aimed at reducing disk space usage of executables. It achieves this by compressing the file's content, and decompressing it during runtime – thus, limiting static analysis methods. As mentioned in [Aghakhani et al., 2020, Mantovani et al., 2020], a significant percentage of samples are packed – also confirmed by our experiments, see Table 6.1.

6.1 Analyzing packed samples

Packed samples may have a negative effect on a classifier based on static features, since its binary code does not reflect the payload's code [Aghakhani et al., 2020, Mantovani et al., 2020] – as we suspected in Section 5.3.8. Therefore, we examined whether correlation exists between packed samples and their family. The correlation was measured using the Cramér's V [Sheskin, 2000] association score between three nominal variables: samples' family, packed status and packer, as shown in Table 6.2.

6.2 Conclusions

This chapter investigated the impact of packers on the classification performance of our proposed model in Section 5.3, and the EMBER vector. We have shown that there is a high correlation between families and packers in three datasets, Mal-

	Total	Packed		UPX	Petite	ASPack	DxPack	MPRESS	PEComp
BODMAS	57 293	18 688	(33%)	9 676	3 795	1 771	1 654	1 174	580
MalImg	9 458	1 703	(18%)	1 686	–	–	–	–	4
IBD	18 756	2 195	(12%)	424	159	111	19	214	909

Table 6.1: Packer statistics obtained by DIE.

	BODMAS	MalImg	IBD
Family – Packed	0.755	0.981	0.736
Family – Packer	0.547	0.590	0.525

Table 6.2: Cramér's V association between malware family, packed status, and packer.

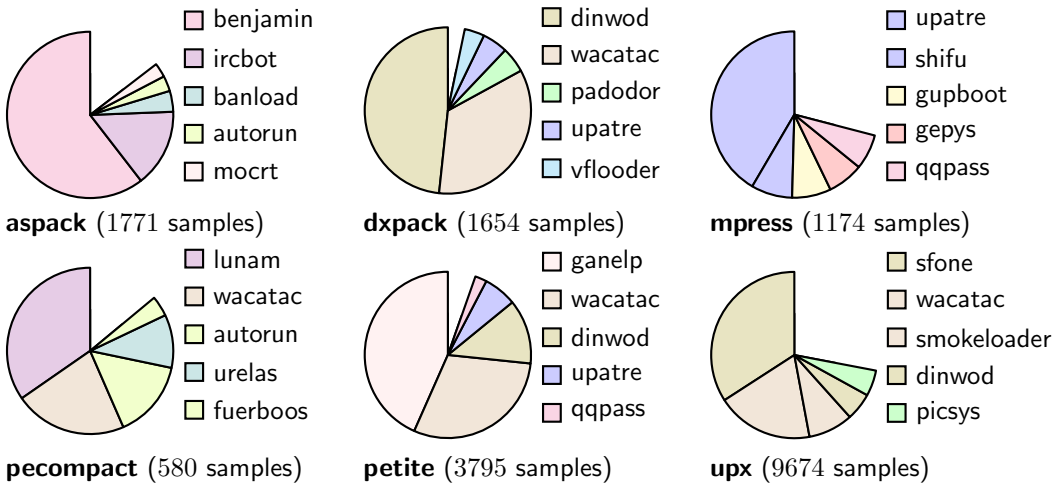


Figure 6.1: Distribution of families for each packer used in BODMAS.

Img, BODMAS and IBD, which span across nearly 15 years and demonstrate matching family-packer distribution in numerous cases – e.g. family *agent* tends to be found non-packed in both BODMAS and IBD, while *upatre* samples are generally not packed, but if packed, then mostly by *mpress* or *petite*. Figure 6.1 shows some interesting cases – *aspack* is predominantly used to pack *benjamin* and *ircbot* families, while *petite* is used to pack *ganelp* and *wacatac* families – in both cases, in $\approx 75\%$ of the samples.

These relevation and can be used as a starting point for further research in this area – since packers can be determined in a relatively programmatic manner, this can be help in narrowing down the potential pool of families to be assigned to the sample.

Chapter 7

Conclusions and future directions

This thesis contributed to the field of malware analysis and attribution (i.e. classification) by exploring various approaches in the literature, providing a comprehensive examination of the techniques and features used to classify malware into families. Each chapter addresses a different method applicable in malware family classification, each of them building on the idea of extracting information from the static call graph of an executable, obtained either with IDA or Radare2 disassembly tools.

Chapter 2 introduces key domain-specific concepts, including types of malware, methods for analysis, and important tools and features essential for investigating malicious files. Chapter 3 focuses on the prominent disassembler tools IDA and Radare2, revealing their scripted functionalities for generating static call graphs. The detailed comparison of call graphs generated by these tools provided valuable insights into their relative strengths and applicability for malware analysis tasks. We also motivate the use of Radare2 in our research due to its fast and reliable python bindings, which facilitate the automation of complex analysis tasks.

Chapter 4 and Chapter 5 present several methodologies for malware family classification – the former uses unsupervised learning techniques, while the latter employs supervised learning. We present a method to cluster malware samples based on locality-sensitive signatures extracted from their call graph. These signatures will generalize the behavior of the sample, being based on the graphs' instructions as well as structure. The clustering is done using community detection algorithms, which are able to group similar samples together. The second approach uses supervised learning to classify malware samples into families. We use graph convolutional networks to learn the topology of the call graph, and then classify the samples using a fully

connected neural network. We also transform the call graphs into images, and apply several state-of-the-art convolutional neural network models to classify the samples. Both approaches have shown high accuracy in classifying malware samples. Future directions for these approaches include the extension of the locality-sensitive hashing method to include more features from the call graph. Another direction is to analyze the robustness of the graph convolutional network model built on the call graph to adversarial attacks, and compare it with other benchmark models, such as the EMBER feature vector. For this analysis however, generating the adversarial samples will be the most challenging part.

Chapter 6 addressed the impact of packers on malware analysis, revealing a strong correlation between the packer used to obfuscate a malicious file and its associated malware family. By analyzing three extensive datasets spanning nearly two decades, we provided a longitudinal perspective on the enduring relationship between packers and malware families, offering valuable insights into this area. Future work is needed to determine reliable methods for detecting packers, as well as to explore the potential for using packer information to enhance malware classification models.

This thesis advances the field of malware analysis by presenting innovative approaches to malware classification, specifically on the level of feature extraction. The methods presented in this thesis have shown high accuracy in classifying malware samples, and can be used to be integrated into malware analysis frameworks. Existing state-of-the-art datasets, BODMAS and MalImg are analyzed by the methods presented in this thesis, and the processed data is published on Kaggle[1] to enhance the transparency of our work and facilitate further research in the field. Our code processing the static call graph of an executable – a core feature used in this thesis – is also publicly available on Github[2].

[1] `https://www.kaggle.com/datasets/amester/malflow`
[2] `https://github.com/attilamester/malflow`

Bibliography

[Aghakhani et al., 2020] Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., and Kruegel, C. (2020). When malware is packin' heat; limits of machine learning classifiers based on static analysis features. In *NDSS*.

[Anderson and Roth, 2018] Anderson, H. S. and Roth, P. (2018). EMBER: an open dataset for training static pe malware machine learning models. arXiv preprint arXiv:1804.04637.

[Andriesse et al., 2016] Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., and Bos, H. (2016). An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium*, pages 583–600.

[Blondel et al., 2008] Blondel, V., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics Theory and Experiment*, 2008:P10008.

[Bruna et al., 2014] Bruna, J., Zaremba, W., Szlam, A., and LeCun, Y. (2014). Spectral networks and deep locally connected networks on graphs. In *International Conference on Learning Representations (ICLR)*.

[Charikar, 2002] Charikar, M. S. (2002). Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388. ACM.

[Chung, 1997] Chung, F. R. (1997). *Spectral Graph Theory*. Number 92 in Regional Conference Series in Mathematics. American Mathematical Society.

[Cohen, 2019] Cohen, I. (2019). Deobfuscating apt32 flow graphs with cutter and radare2. Technical report, Check Point Software Technologies.

[Cui et al., 2018] Cui, Z., Xue, F., Cai, X., Cao, Y., Wang, G.-g., and Chen, J. (2018). Detection of malicious code variants based on deep learning. *IEEE Transactions on Industrial Informatics*, 14(7):3187–3196.

[Cunningham et al., 2019] Cunningham, E., Boydell, O., Doherty, C., Roques, B., and Le, Q. (2019). Using text classification methods to detect malware. In *AICS*.

[Dahl et al., 2013] Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. (2013). Large-scale malware classification using random projections and neural networks. In *ICASSP*, pages 3422–3426. IEEE.

[Dam and Touili, 2017] Dam, K.-H.-T. and Touili, T. (2017). Malware detection based on graph classification. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy, SCITEPRESS-Science and Technology Publications*.

[de Oliveira and Sassi, 2021] de Oliveira, A. S. and Sassi, R. J. (2021). Behavioral malware detection using deep graph convolutional neural networks. *International Journal of Computer Applications*, 174.

[Dener and Gulburun, 2023] Dener, M. and Gulburun, S. (2023). Clustering-aided supervised malware detection with specialized classifiers and early consensus. *Computers, Materials & Continua*, 75:1235–1251.

[Deng et al., 2023] Deng, H., Guo, C., Shen, G., Cui, Y., and Ping, Y. (2023). MCTVD: A malware classification method based on three-channel visualization and deep learning. *Computers & Security*, 126:103084.

[Feng et al., 2014] Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: semantics-based detection of android malware through static analysis. In *SIGSOFT*, pages 576–587. ACM.

[Fortunato, 2010] Fortunato, S. (2010). Community detection in graphs. *Physics Reports*, 486(3-5):75–174.

[Fu et al., 2018] Fu, J., Xue, J., Wang, Y., Liu, Z., and Shan, C. (2018). Malware visualization for fine-grained classification. *IEEE Access*, 6:14510–14523.

[Gao et al., 2022] Gao, Y., Hasegawa, H., Yamaguchi, Y., and Shimada, H. (2022). Malware detection by control-flow graph level representation learning with graph isomorphism network. *IEEE Access*, 10:111830–111841.

[Gibert et al., 2020] Gibert, D., Mateu, C., and Planes, J. (2020). The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications*, 153:102526.

[Gibert et al., 2019] Gibert, D., Mateu, C., Planes, J., and Vicens, R. (2019). Using convolutional neural networks for classification of malware represented as images. *Journal of Computer Virology and Hacking Techniques*, 15:15–28.

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT Press.

[Guide, 2011] Guide, P. (2011). Intel® 64 and ia-32 architectures software developer's manual. *Volume 3B: System programming Guide, Part*, 2(11):0–40.

[Hai et al., 2023] Hai, T. H., Thieu, V. V., Duong, T. T., Nguyen, H. H., and Huh, E. N. (2023). A proposed new endpoint detection and response with image-based malware detection system. *IEEE Access*, 11:122859–122875.

[Harang and Rudd, 2020] Harang, R. and Rudd, E. M. (2020). Sorel-20m: A large scale benchmark dataset for malicious pe detection. arXiv preprint arXiv:2012.07634.

[Hassen and Chan, 2017] Hassen, M. and Chan, P. K. (2017). 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, Scottsdale, AZ, USA. ACM.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *CVPR*, pages 770–778.

[Henaff et al., 2015] Henaff, M., Bruna, J., and LeCun, Y. (2015). Deep convolutional networks on graph-structured data. arXiv preprint arXiv:1506.05163.

[Hong et al., 2018] Hong, J., Park, S., Kim, S.-W., Kim, D., and Kim, W. (2018). Classifying malwares for identification of author groups. *Concurrency and Computation: Practice and Experience*, 30(3):e4197.

[Hong et al., 2019] Hong, J., Park, S.-J., Kim, T., Noh, Y.-K., Kim, S.-W., Kim, D., and Kim, W. (2019). Malware classification for identifying author groups: a graph-based approach. In *RACS*, pages 169–174. ACM.

[Hong et al., 2020] Hong, S., Xu, Y., Khare, A., Priambada, S., Maher, K., Aljiffry, A., Sun, J., and Tumanov, A. (2020). Holmes: Health online model ensemble serving for deep learning models in intensive care units. In *SIGKDD*, pages 1614–1624.

[Howard et al., 2019] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B., Tan, M., Wang, W., Zhu, Y., Pang, R., Vasudevan, V., et al. (2019). Searching for mobilenetv3. In *ICCV*, pages 1314–1324.

[Huang et al., 2017] Huang, G., Liu, Z., Van Der Maaten, L., and Weinberger, K. Q. (2017). Densely connected convolutional networks. In *CVPR*, pages 4700–4708.

[Hussain et al., 2022] Hussain, M. J., Shaoor, A., Baig, S., Hussain, A., and Muqurrab, S. A. (2022). A hierarchical based ensemble classifier for behavioral malware detection using machine learning. In *IBCAST*, pages 702–706.

[Jia et al., 2023] Jia, L., Yang, Y., Tang, B., and Jiang, Z. (2023). ERMDS: A obfuscation dataset for evaluating robustness of learning-based malware detection system. *BenchCouncil Transactions on Benchmarks, Standards and Evaluations*, 3:100106.

[Jiang et al., 2018] Jiang, H., Turki, T., and Wang, J. T. (2018). DLGraph: Malware detection using deep learning and graph embedding. In *ICMLA*, pages 1029–1033. IEEE.

[Kalash et al., 2018] Kalash, M., Rochan, M., Mohammed, N., Bruce, N. D., Wang, Y., and Iqbal, F. (2018). Malware classification with deep convolutional neural networks. In *NTMS*, pages 1–5. IEEE.

[Kilgallon et al., 2017] Kilgallon, S., De La Rosa, L., and Cavazos, J. (2017). Improving the effectiveness and efficiency of dynamic malware analysis with machine learning. In *2017 Resilience Week (RWS)*, pages 30–36.

[Kim et al., 2023] Kim, J., Paik, J. Y., and Cho, E. S. (2023). Attention-Based Cross-Modal CNN Using Non-Disassembled Files for Malware Classification. *IEEE Access*, 11:22889–22903.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. arXiv preprint arXiv:1609.02907.

[Koo et al., 2021] Koo, H., Park, S., and Kim, T. (2021). A look back on a function identification problem. In *Annual Computer Security Applications Conference*, pages 158–168.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *NeurIPS*.

[Li et al., 2021] Li, S., Zhou, Q., Zhou, R., and Lv, Q. (2021). Intelligent malware detection based on graph convolutional network. *The Journal of Supercomputing*, pages 1–17.

[Maillet and Marais, 2023] Maillet, W. and Marais, B. (2023). Neural networks optimizations against concept and data drift in malware detection. arXiv preprint arXiv:2308.10821.

[Mantovani et al., 2020] Mantovani, A., Aonzo, S., Ugarte-Pedrero, X., Merlo, A., and Balzarotti, D. (2020). Prevalence and impact of low-entropy packing schemes in the malware ecosystem. In *NDSS*.

[Martinelli et al., 2017] Martinelli, F., Marulli, F., and Mercaldo, F. (2017). Evaluating convolutional neural network for effective mobile malware detection. *Procedia Computer Science*, 112:2372–2381.

[Massarelli et al., 2019] Massarelli, L., Di Luna, G. A., Petroni, F., Baldoni, R., and Querzoni, L. (2019). Safe: Self-attentive function embeddings for binary similarity. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 309–329, Cham. Springer International Publishing.

[McLaughlin et al., 2017] McLaughlin, N., Martinez del Rincon, J., Kang, B., et al. (2017). Deep Android malware detection. In *CODASPY*, pages 301–308. ACM.

[Mester, 2020] Mester, A. (2020). Scalable, real-time malware clustering based on signatures of static call graph features. Master's thesis, Babeş–Bolyai University, Faculty of Mathematics and Computer Science, Cluj-Napoca, Romania.

[Mester, 2023] Mester, A. (2023). Malware analysis and static call graph generation with Radare2. *Studia Universitatis Babeș-Bolyai Informatica*, 68(1):5–20.

[Mester et al., 2025] Mester, A., Bodó, Z., Vinod, P., and Conti, M. (2025). Towards a malware family classification model using static call graph instruction visualization. In *Network and System Security*, pages 167–186, Singapore. Springer Nature Singapore.

[Mester and Bodó, 2021] Mester, A. and Bodó, Z. (2021). Validating static call graph-based malware signatures using community detection methods. In *ESANN*, pages 429–434.

[Mester and Bodó, 2022] Mester, A. and Bodó, Z. (2022). Malware classification based on graph convolutional neural networks and static call graph features. In *IEA/AIE*, pages 528–539. Springer.

[Mester et al., 2021] Mester, A., Pop, A., Mursa, B.-E.-M., Greblă, H., Dioşan, L., and Chira, C. (2021). Network analysis based on important node selection and community detection. *Mathematics*, 9(18):2294.

[Moussas and Andreatos, 2021] Moussas, V. and Andreatos, A. (2021). Malware detection based on code visualization and two-level classification. *Information*, 12:1–14.

[Nar et al., 2019] Nar, M., Kakisim, A. G., Yavuz, M. N., and Soğukpinar, İ. (2019). Analysis and comparison of disassemblers for opcode based malware analysis. In *2019 4th International Conference on Computer Science and Engineering (UBMK)*, pages 17–22. IEEE.

[Nataraj et al., 2011] Nataraj, L., Karthikeyan, S., Jacob, G., and Manjunath, B. S. (2011). Malware images: visualization and automatic classification. In *VizSec*, pages 1–7.

[Ni et al., 2018] Ni, S., Qian, Q., and Zhang, R. (2018). Malware identification using visualization images and deep learning. *Computers & Security*, 77:871–885.

[Oprisa et al., 2014] Oprisa, C., Checiches, M., and Nandrean, A. (2014). Locality-sensitive hashing optimizations for fast malware clustering. In *Proceedings of the 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*, pages 97–104, Cluj-Napoca, Romania. IEEE.

[Park et al., 2010] Park, Y., Reeves, D., Mulukutla, V., and Sundaravel, B. (2010). Fast malware classification by automated behavioral graph matching. In *CSIIRW*, pages 1–4.

[Phan et al., 2018] Phan, A. V., Le Nguyen, M., Nguyen, Y. L. H., and Bui, L. T. (2018). DGCNN: A convolutional neural network over large-scale labeled graphs. *Neural Networks*, 108:533–543.

[Priyanga et al., 2022] Priyanga, S., Suresh, R., Romana, S., and Shankar Sriram, V. (2022). The good, the bad, and the missing: A comprehensive study on the rise of machine learning for binary code analysis. In *Computational Intelligence in Data Mining: Proceedings of ICCIDM 2021*, pages 397–406. Springer.

[Quertier et al., 2022] Quertier, T., Marais, B., Morucci, S., and Fournel, B. (2022). MERLIN – Malware Evasion with Reinforcement LearnINg. arXiv preprint arXiv:2203.12980.

[Radare, 2009] Radare (2009). The official radare2 book. https://book.rada.re/.

[Ronen et al., 2018] Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., and Ahmadi, M. (2018). Microsoft malware classification challenge. arXiv preprint arXiv:1802.10135.

[Sandor et al., 2023] Sandor, M., Portase, R. M., and Colesa, A. (2023). Ember feature dataset analysis for malware detection. In *ICCP*, pages 203–210.

[Shaila et al., 2021] Shaila, S., Darki, A., Faloutsos, M., Abu-Ghazaleh, N., and Sridharan, M. (2021). Disco: Combining disassemblers for improved performance. In *Proceedings of the 24th International Symposium on Research in Attacks, Intrusions and Defenses*, pages 148–161.

[Sheskin, 2000] Sheskin, D. J. (2000). *Handbook of parametric and nonparametric statistical procedures*. Chapman & Hall/CRC.

[Simonyan and Zisserman, 2015] Simonyan, K. and Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

[Steffens, 2020] Steffens, T. (2020). *Attribution of Advanced Persistent Threats*. Springer.

[Szegedy et al., 2015] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *CVPR*, pages 1–9.

[Tahir, 2018] Tahir, R. (2018). A study on malware and malware detection techniques. *International Journal of Education and Management Engineering*, 8(2):20.

[Tan and Le, 2021] Tan, M. and Le, Q. (2021). EfficientNetV2: Smaller models and faster training. In *ICML*, pages 10096–10106. PMLR.

[Tang and Qian, 2019] Tang, M. and Qian, Q. (2019). Dynamic api call sequence visualisation for malware classification. *IET Information Security*, 13(4):367–377.

[Topan et al., 2013] Topan, V. I., Dudea, S. V., and Canja, V. D. (2013). Fuzzy whitelisting anti-malware systems and methods. US Patent 8,584,235.

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

[Wenzl et al., 2019] Wenzl, M., Merzdovnik, G., Ullrich, J., and Weippl, E. (2019). From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37.

[Wu et al., 2019] Wu, F., Souza, A., Zhang, T., Fifty, C., Yu, T., and Weinberger, K. (2019). Simplifying graph convolutional networks. In *International Conference on Machine Learning*, pages 6861–6871. PMLR.

[Xiao et al., 2021] Xiao, M., Guo, C., Shen, G., Cui, Y., and Jiang, C. (2021). Image-based malware classification using section distribution information. *Computers & Security*, 110:102420.

[Yan et al., 2022] Yan, J., Jia, X., Ying, L., Yan, J., and Su, P. (2022). Understanding and mitigating label bias in malware classification: An empirical study. In *QRS*, pages 492–503.

[Yan et al., 2019] Yan, J., Yan, G., and Jin, D. (2019). Classifying malware represented as control flow graphs using deep graph convolutional neural network. In *DSN*, pages 52–63. IEEE.

[Yang et al., 2021] Yang, L., Ciptadi, A., Laziuk, I., Ahmadzadeh, A., and Wang, G. (2021). BODMAS: An Open Dataset for Learning based Temporal Analysis of PE Malware. In *SPW*, pages 78–84.

[Yin et al., 2018] Yin, X., Liu, S., Liu, L., and Xiao, D. (2018). Function recognition in stripped binary of embedded devices. *IEEE Access*, 6:75682–75694.

[Yuan et al., 2020] Yuan, B., Wang, J., Liu, D., Guo, W., Wu, P., and Bao, X. (2020). Byte-level malware classification based on markov images and deep learning. *Computers & Security*, 92:101740.

[Zhan et al., 2023] Zhan, D., Duan, Y., Hu, Y., Yin, L., Pan, Z., and Guo, S. (2023). AMGmal: Adaptive mask-guided adversarial attack against malware detection with minimal perturbation. *Computers & Security*, 127.

[Zhou et al., 2020] Zhou, J., Cui, G., Hu, S., Zhang, Z., Yang, C., Liu, Z., Wang, L., Li, C., and Sun, M. (2020). Graph neural networks: A review of methods and applications. *AI Open*, 1:57–81.