ČVUT ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kouba** Jméno: **Dominik** Osobní číslo: **466040**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Kybernetická bezpečnost**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Analýza záznamů běhu malwaru pomocí hierarchického multi-instančního učení

Název diplomové práce anglicky:

Analyzing the execution of malware in a sandbox using hierarchical multiple instance learning

Pokyny pro vypracování:

The thesis aims to capture and analyze artifacts of malware execution in a protected environment and assess if these artifacts can be used to predict malware functionalities and capabilities.
1. Run several instances of CapeV2 sandbox and solve their orchestration for this experiment
2. Capture behavior of selected malware samples in CapeV2 sandbox and store results
3. Learn the hierarchical multiple instance learning framework (HMill)
4. Analyze captured data. Report basic statistics and choose appropriate features and hidden states for further modeling.
5. Using HMill, create models, and identify the artifacts corresponding to different malware behavior. Report results.
6. Investigate which parts of the CapeV2 log are important to different malware behavior.
7. Evaluate the results of the experiment.

Seznam doporučené literatury:

1. Jan Stiborek, Tomáš Pevný, and Martin Rehák. „Multiple instance learning for malware classification“ Expert Syst. Appl. 93, C (March 2018), 346–357, 2018.
2. Digit Oktavianto and Iqbal Muhardianto. „Cuckoo Malware Analysis“. Packt Publishing, 2013.
3. T. Pevný and P. Somol, „Using neural network formalism to solve multiple-instance problems,” in International Symposium on Neural Networks, pp. 135–142, Springer, 2017.
4. S. Mandlik, „Mapping the Internet — Modelling Entity Interactions in Complex Heterogeneous Networks (diploma thesis)“, 2020.
5. Wang C., Ding J., Guo T., Cui B. „A Malware Detection Method Based on Sandbox, Binary Instrumentation and Multidimensional Feature Extraction“. In: Barolli L., Xhafa F., Conesa J. (eds) Advances on Broad-Band Wireless Computing, Communication and Applications. BWCCA 2017. Lecture Notes on Data Engineering and Communications Technologies, vol 12. Springer, Cham., 2018.

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Tomáš Pevný, Ph.D., centrum umělé inteligence FEL

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

Datum zadání diplomové práce: **11.02.2021** Termín odevzdání diplomové práce: ______________

Platnost zadání diplomové práce: **30.09.2022**

doc. Ing. Tomáš Pevný, 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

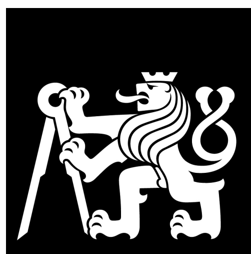
© VUT v Praze, Design: VUT v Praze, VIC

Analyzing the execution of malware in a sandbox using hierarchical multiple instance learning

Master's Thesis

Bc. Dominik Kouba

Supervisor: doc. Ing. Tomáš Pevný, Ph.D.



FACULTY
OF ELECTRICAL
ENGINEERING
CTU IN PRAGUE

Department of Computer Science

Aknowledgements

My deepest gratitude goes to my supervisor, doc. Ing. Tomáš Pevný, Ph.D. for giving me advice and helping me overcome struggles during this journey. I also sincerely thank doc. Mgr. Branislav Bošanský, Ph.D., Ing. Thorsten Sick and Ing. Josef Liška for their advice and encouragement. I would like to acknowledge my friends Denis and Hoang for their wonderful collaboration on proofreading. Finally, I wish to extend my special thanks to my family and girlfriend Terezka for their support during my studies.

Computational resources were supplied by the project "e-Infrastruktura CZ" (e-INFRA LM2018140) provided within the program Projects of Large Research, Development and Innovations Infrastructures. This support was very beneficial.

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.

In Prague on May 15, 2021 ....

Abstract

The goal of the thesis was to perform a dynamic malware analysis of portable executable files and create a dataset of analysis reports. Furthermore, we aimed at statistical modelling of the collected data and evaluation of models. Lastly, we intended to explain the model's predictions using statistical methods and discuss the results. The whole process was motivated by model interpretability to achieve greater compliance of machine learning and cybersecurity. We used *CAPEv2* sandbox for malware analysis and implemented a pipeline that downloads malware samples, distributes them among multiple sandbox instances, and finally collects results. The Hierarchical Multiple Instance Learning (*HMill*) framework was used to model the dependence of malware signatures on behavioural features like API calls, dropped files and processes, both presented in analysis *JSON* report. The HMill framework uses a tree-structured neural net to model the structure of the input document. Trained models were explained using *Banzhaf* values and minimal subtree selection. We analyzed 80,000 publicly accessible malware samples and collected results, from which signatures and behavioural features were extracted. A binary classifier was trained for each extracted signature. Nine out of twelve signatures resulted in a balanced accuracy above 90%, which was sufficient for model explanation experiments. Even though there might be hundreds of entries in the original behavioural report, the explainer only provided 3–5 entries as an explanation for each model. To evaluate the explanation, Python implementation of the signatures was examined to get their true cause. It is evident from our observations that some models are intensely associated with the original signature's cause. It is worth noting that there are cases where the model used different behavioural features with high accuracy.

Keywords: *Multilple instance learning, cybersecurity, malware*

Abstrakt

Cílem této práce bylo provést dynamickou analýzu spustitelných souborů a vytvořit datovou sadu reportů. Dále jsme se zaměřili na statistické modelování shromážděných dat a vyhodnocení přesnosti těchto modelů. Nakonec jsme použili statistické metody pro vysvětlení predikcí a diskutovali výsledky. Motivace experimentu byla převážně kvalita interpretace modelu a její přínos k dosažení spolupráce strojového učení a kybernetické bezpečnosti. Pro analýzu malwaru jsme použili sandbox *CAPEv2* a implementovali jsme celý proces od stažení vzorků až po sběr výsledků analýz. Modelování bylo realizováno pomoci frameworku hierarchického multi-instančního učení (*HMill*). Vstupními vektory pro model byly záznamy o chování (např. volání API, uložené soubory, procesy) a výstupními třídami byly detekované znaky chování (tzv. *signature*), obojí k dispozici v získaných /emphJSON reportech. *HMill* používá pro modelování vstupního dokumentu stromově strukturovanou neuronovou síť. Natrénované modely byly vysvětleny pomocí *Banzhafových* hodnot a metody výběru minimálního podstromu. Zanalyzovali jsme 80,000 veřejně přístupných vzorků malwaru a shromáždili výsledky, ze kterých byly získány vstupní vektory a výstupní třídy pro náš model. Pro každý *signature* byl natrénován binární klasifikátor. U devíti z celkových dvanácti modelů jsme pozorovali přesnost nad 90%, což bylo dostatečné pro následující experimenty s vysvětlováním modelu. Přestože původní report může obsahovat i stovky položek, vysvětlení pro každý model obsahovalo pouze 3–5 položek. Abychom byli schopni lépe vyhodnotit vysvětlení modelů, prozkoumali jsme Python implementaci každého *signature* a nalezli jeho skutečnou příčinu. Z našich pozorování je zřejmé, že některé modely do svých predikcí zapojují původní příčiny. Zároveň stojí za zmínku, že některé modely s vysokou přesností použili i jiné části původního vstupního vektoru.

Keywords: *Multiinstanční hierarchické učení, kyberbezpečnost, malware*

Contents

1 Introduction **1**

I Theory and prior **5**

2 Malware analysis **7**
2.1 Malware . . . 7
2.2 File formats . . . 8
2.2.1 Portable Executable . . . 8
2.3 Malware analysis . . . 9
2.3.1 Static analysis . . . 10
2.3.2 Dynamic analysis . . . 11
2.4 CAPEv2 . . . 13
2.4.1 Architecture . . . 13
2.4.2 Components . . . 13
2.4.3 Analysis flow . . . 16
2.4.4 Network setup options . . . 16
2.4.5 Other features . . . 16
2.4.6 Produced data . . . 17
2.4.6.1 Report . . . 17
2.5 Prior arts . . . 18

3 Malware classification **20**
3.1 Machine learning . . . 20
3.1.1 Definitions . . . 20
3.1.2 Machine learning tasks . . . 22
3.2 Loss functions for classification . . . 23
3.3 Model Evaluation . . . 24
3.4 Neural Networks . . . 25
3.5 Tree-structured data classification . . . 26
3.5.1 Rules . . . 27
3.5.2 Flattening . . . 27
3.5.3 Graph neural networks . . . 27
3.5.4 HMill framework . . . 28

4 Hierarchical Multiple Instance learning **29**
4.1 Multiple instance learning . . . 29
4.2 Mill problem solution . . . 30
4.2.1 Instance-Space paradigm . . . 30
4.2.1.1 Standard assumption . . . 31
4.2.1.2 Collective assumption . . . 31
4.2.2 Bag-Space paradigm . . . 31
4.2.3 Embedded-Space paradigm . . . 32
4.2.3.1 Vocabulary-based methods . . . 32
4.2.3.2 Histogram-based methods . . . 32
4.3 HMill framework . . . 33
4.3.1 Abstract data nodes . . . 33
4.3.1.1 Array node . . . 33
4.3.1.2 Bag node . . . 34
4.3.1.3 Product node . . . 34
4.3.2 HMill schema . . . 34
4.3.3 HMill model . . . 35
4.3.3.1 Array model — $am(f)$ . . . 35
4.3.3.2 Bag model — $bm(f, g, F)$ . . . 36
4.3.3.3 Product model — $pm(f_1, \ldots, f_l, f_p)$ . . . 36
4.3.4 Modelling JSON documents using HMill framework . . . 36
4.4 CAPEv2 classification . . . 37
4.4.1 Behavioural features . . . 37
4.4.2 Signature classes . . . 38
4.4.2.1 Categorization . . . 38
4.4.3 Model . . . 38

5 Model explaining **40**
5.1 Definition . . . 40
5.2 Categorization . . . 41
5.3 Explanability desiderata . . . 42
5.3.1 Trust . . . 42
5.3.2 Causality X Correlation . . . 42
5.3.3 Transferability . . . 43
5.3.4 Informativness . . . 43
5.3.5 Fair and ethical decision making . . . 43
5.4 Interpretable model . . . 43
5.4.1 Transparency . . . 43
5.4.2 Post-hoc . . . 44
5.5 Explaining HMill models . . . 45
5.5.1 Explainer steps . . . 45
5.5.2 Subset selection . . . 46
5.5.3 Minimal subtree adaptation . . . 46
5.5.4 Subtree ranking . . . 46
5.5.4.1 Model gradient ranking . . . 46
5.5.4.2 GNN explainer mask ranking . . . 47

5.5.4.3 Banzhaf values 47
5.5.5 Results 47
5.6 Other methods for structured data 47
5.7 CAPEv2 explaining 48
5.7.1 Additions 48

II Experiments 49

6 Infrastucture and data collection 51
6.1 Host machine 51
6.2 Guest machine 52
6.3 Network setup 52
6.3.1 None 53
6.3.2 Internet 53
6.4 Data collection pipeline 54
6.4.1 Abuse.ch MalwareBazaar 54
6.4.2 File filtering 55
6.4.3 Metadata 55
6.4.4 Distributed sandbox 55
6.4.5 Result postprocessing 55
6.5 Collected dataset 55

7 Results 57
7.1 Model 57
7.1.1 Details 57
7.1.1.1 Hyperparameters 57
7.1.1.2 Experiments 58
7.1.1.3 Technicalities 58
7.1.2 Results 58
7.1.3 Discussion 58
7.2 Explainer 60
7.2.1 Details 60
7.2.2 Results 60
7.2.3 Discussion 60

8 Conclusions 64

A Classifier evaluation metrics 66

B CAPEv2 sandbox details 68

C Network architecture for distributed sandbox 71

D Signatures description 73

E Model metrics 76

F Explaining details **81**

G Technology **82**

H Attachments **84**

List of Figures

2.1 PE file structure [39] . . . 9
2.2 Official image of sandbox architecture [15] . . . 14
2.3 *CAPEv2* components and analysis flow [99] . . . 17

3.1 Neural net example . . . 26

4.1 Supervised learning . . . 30
4.2 Hierarchical *Mill* model by [83] (image inspired by [68]) . . . 33
4.3 Representation of a plant specimen in *HMill* framework [68] . . . 34
4.4 Translation of *JSON* document into *HMill* data nodes [68] . . . 37

C.1 *None* network setup . . . 71
C.2 *Internet* network setup . . . 72

E.1 antidebug setunhandledexceptionfilter plots . . . 77
E.2 copiesself plots . . . 77
E.3 deletesself plots . . . 77
E.4 enumerates running processes plots . . . 78
E.5 stealthtimeout plots . . . 78
E.6 uses windows utilities plots . . . 78
E.7 removeszoneidads plots . . . 79
E.8 antisandboxsleep plots . . . 79
E.9 dropper plots . . . 79
E.10 invalid authenticode signature plots . . . 80
E.11 packer entropy plots . . . 80
E.12 stealth network plots . . . 80

List of Tables

2.1 CAPEv2 Sandbox output 18

3.1 Confusion matrix for binary classifier 24

7.1 Hyperparameters of *HMill* model 57
7.2 Model performance (values are rounded off to three decimal digits, P denotes positive examples ratio in our dataset) 59

A.1 Classifier evaluation metrics 67

B.1 Parts of *report.json* 69
B.2 Behaviour parts of *report.json* 70

D.1 Used signatures and their details 75

E.1 Additional classifier results for particular signatures (rounded off to 3 decimal digits) 76

F.1 Details for each signature's model explanation 81

Chapter 1

Introduction

Motivation

In this day and age, we face an intense explosion of machine learning applications in various branches of human efforts such as biology, chemistry, physics, and others. These technologies widely influence our daily lives and make them immensely more convenient, faster, and more enjoyable. On the other hand, there are many cases where algorithms (especially in machine learning) can control our decisions, reasoning, and life.

If we use these computer science tools appropriately, we can often create something that may serve our protection. We can take the detection of threats and frauds in cybersecurity as a perfect example, research and applications in this particular field are fascinating for multiple reasons, and one of them is our motivation. We have to know which side we are standing on and what the interests of our clients are. In the case of fraud detection, we know that the investment is profitable only if the fraud has a significant financial impact. Not all frauds are interesting from a financial point of view because solving them also costs much money. However, from an ethical point of view, every fraud should be punished. Similarly, network security, single device security, and access control are often disregarded. Small businesses targeting a specific market are not interested in costly services whose impact is mainly preventive. The primary objective of such business is cost reduction and financial profit. Nevertheless, it cannot be denied that loss of privacy and data is undesirable. We have to start analyzing costs, benefits, risks, probability, and impact (potential damage). That is not so evident in a technical branch as cybersecurity is.

An inseparable character of this play is malware. Let us motivate this thesis by listing several examples.

Firstly, one of the most prevalent malicious software is *ransomware*. Its overall damage is estimated to be $20 billion, increasing every year [85]. Though the social impact might be arguably even more significant than the financial side as there have been attacks targetting even healthcare organizations, which is further exacerbated by the fact that the first death following a ransomware attack was reported in 2020 [21].

We can conclude that IoT malware is becoming more common, supported by Sonic Wall's 2019 report. That is caused by the insufficient protection of these small devices, for which we cannot provide complete malware protection. However, 127 new IoT devices are being

connected to the internet every second, which leads us to an estimation that by the end of 2021, there will be 35 billion IoT devices connected to the internet [67]. This risk can not be mitigated easily, and malware elimination will play an even more significant role as it has so far.

Another convenient trend for malware is widespread encryption, which has become a standard in web traffic. Its main goal is the security of information. Considering this fact, the creators of malware have a lot to hide and secure from the protectors as well. The encryption might inform us that the source has something that nobody else should see. A long-lasting trend of such behaviour might be suspicious. We can check if there is a justified reason to encrypt the data, or we can at least make some conclusions about the source of the encrypted data. Nevertheless, that is not possible in the world where everything is encrypted.

In 2020, 94% of malware was delivered by email [33], and therefore the importance of phishing emails with malicious attachments and other social engineering techniques grows. It is cheaper to produce one sophisticated, convincing email to retrieve some information than an attempt to attack a highly protected network perimeter. It also might be used to distribute malware or other threats.

In 2020 AV-atlas [98] recorded over 750 million malware samples, and moreover, at the end of April 2021, this number has increased to over 820 million malware samples. The majority of them are executable files attacking Windows devices.

Malware research continues, and it will undoubtedly do so until we can introduce a sufficiently universal and flexible solution that will be able to detect zero-day threats (unseen). We might find a solution among machine learning models, which are often involved. However, its challenge is interpretability and explainability, not only in cybersecurity. We face the problem that a model's performance is often significant, but we are not sure about the reason, and it is risky to deploy such a model to a situation where it can meet unseen data. High-quality security engineers do not have to be high-quality machine learning engineers. If we want to involve machine learning methods increasingly, we need to interpret and explain its predictions to combine cybersecurity knowledge with the results of the models and gain a better understanding.

Goal

The main objective of this thesis is to design a pipeline that has a malware file dataset as the input and a machine learning model and its explanation as the output. We want to go through the whole process, document each step, and report results. Our acquisition is the process itself, so it is described in detail for the reader to identify the problems we experienced and replicate or extend our setup.

From the assignment of this thesis, we can extract the following steps:

1. Run several instances of *CAPEv2* [76] sandbox and solve their orchestration for this experiment
2. Capture the behaviour of selected malware samples and store the results

3. Learn the hierarchical multiple instance learning (*HMill*) framework
4. Analyze the captured data. Report the basic statistics and choose appropriate features and hidden states for further modelling
5. Using HMill, create models and identify the artefacts corresponding to different malware behaviour, and report the results
6. Investigate which parts of the *CAPEv2* log are important to different malware behaviour

In detail

The first step implies using dynamic malware analysis to retrieve the input for our machine learning model. This intention originated after we downloaded a couple of thousands of sandbox *JSON* reports from the internet and examined them. We observed that this use case might be challenging for our method and might demonstrate its capabilities.

The initial task is data collection. We are about to use MalwareBazaar[1] as a public data source of malware samples. We chose it because of its free access with no claims for usage and a reasonable amount of samples. We aim at *Portable Executable* (PE), which does not require any additional software running on the target machine. The sandbox we want to use is *CAPEv2* [76] because the first reports were also produced by this tool, and they are sufficient for analysis purposes. It is a fork of a popular *Cuckoo* sandbox which is no longer maintained. The sandbox has to be run in multiple instances to collect a sufficient number of samples.

The model we want to use is a *hierarchical multiple instance learning* model. In [68], the authors showed that this model has good performance modelling *JSON* documents. After further research, we decided to model the dependence of malware *signatures* on behavioural features, both included in *JSON* report. Signatures are the essential input for the original classification techniques used by the sandbox, and we want to see how well the model predicts them based on malware behaviour.

Finally, we will attempt to explain the predictions by choosing a minimal subset of features that contributes to the model's prediction the most. The explanation will be performed using the existing *HMill* explainer. We can study the implementation of signatures and their true cause, which might help us with results evaluation. We expect that explanation of the model, the cause of which is in the report, should be a set containing this cause. As an example, we expect that if the original signature's cause is a specific API call, it should be presented in the explanation of the binary classifier for this signature. As authors of *HMill* explainer mentioned, we hope that the explanation contains even something new. In other words, we expect that we could observe explanations that contain entries that are not the original cause. However, they might reasonably substitute it — they are connected to the same effects. It is also possible to identify new signatures because all samples are malicious and the model might generalize based on different similarities in the training set.

[1] https://bazaar.abuse.ch/

Thesis structure

The thesis is divided into two main parts. In the first part, we focus on the theoretical background of our method. In the second part, we present a specific setup, our results, and their discussion. More complex structures (images, tables) are part of the appendices. The attachments containing additional material are described in H.

The theoretical part starts with the malware analysis theory in chapter 2 where we break down the malware itself, the types of its analyses, and its output. We continue in the chapter 3 which describes the machine learning formalism, cybersecurity context, and structured data (*JSON* document) usage in machine learning. Finally, the chapters 4 and 5 describe the particular methods used in our modelling and explaining experiments.

The second part consists of two chapters. Chapter 6 includes a description of the infrastructure and the data collection process. Chapter 7 presents the model and explainer setup, results, and their discussion.

Part I

Theory and prior

Chapter 2

Malware analysis

A particular goal of this thesis is to create a dataset of dynamic malware analysis samples. In this chapter, we define the basic notions of malware, PE file and malware analysis. We describe *CAPEv2* sandbox and its output.

2.1 Malware

Malicious code (also called *Malware*) is intended to disrupt a computer, network, or its parts from functioning. The form or format may vary. It could be a JAVA application, Microsoft Office macro or even a PDF file. Malware detection, classification and overall research is part of the *Intrusion detection* [23]. Attackers use malware to steal data, use target computer (in C2 or DDOS attacks) or even spy on the owner. Malware often gets into the target computer via usual communication channels: email, flash drive, downloaded. [53]. Malware may contain more parts that have a different role, and we call these parts *components*. The software which is not malicious (benign) is often called *cleanware*.

Firstly, let us clarify several terms. *Malware type* denotes a group of malware samples or their components that show the same behaviour. *Malware family* is a collection of malware that has the same code base (uses same code components). An example of such a family might be *Emotet*, a banking trojan family, or CryptoWall, a ransomware family. *Malware variant/strain/version* is malware that belongs to some existing family but includes new parts which were not earlier detected in this family [22].

Based on the specific behaviour, we can distinguish several fundamental types of malware. Each malware sample may have multiple components, and each component may have a different type. The most usual malware types follow (their source is [23, 53, 41, 93]).

- *Worm* is malware capable of copying itself in a variety of ways and spreading on multiple devices.
- *Virus* is capable of infecting other executable files by injecting its payload. The user then executes the infected files.
- *Trojan* disguises itself as a regular program. After installation, they can steal the data, control the target machine, etc.

- *Backdoor* allow the attacker to access the target machine. The attacker can use multiple machines as a botnet controlled by a command and control server.
- *Adware* presents unwanted advertisements to a user of the target machine. They are usually downloaded from the internet.
- *Information stealer* aims at the user's data. Examples are sniffers, grabbers, spyware, and key loggers.
- *Ransomware* locks users out of their computer or encrypts their data. It usually threatens the user by not decrypting it before they pay some money.
- *Rootkit* allows a malware presence concealment.
- *Dropper* downloads a malicious code or its update from the internet. The dropper itself is often harmless.
- *Launcher* is responsible for running malicious code, usually stealthy.

According to [98], the most seen types on Windows operating system are *trojans* and *backdoors*.

2.2 File formats

Malware is defined very generally, and it is not limited to a specific file format. We can find various file types — Portable executable, Portable Document Format, Microsoft office formats, HTML, Archives and many others. According to [98] Portable Executable (PE) files are the most seen on Windows machines, and therefore this thesis exclusively focuses on it.

2.2.1 Portable Executable

The portable executable is a file format mainly for executables, object code and data-link libraries (DLLs). It is specific for Windows operation system. For the 32-bit version, we use the format PE32 and for the 64-bit version is PE32+. Microsoft provides its complete description in [70]. Description of the essential parts follows.

The PE file includes information necessary for Windows operating system (specifically the dynamic linker) to map the file into memory [39]. Usual file extensions are *.exe, .dll, .sys*. It consists of two main parts — *Header* and *Sections* (see 2.1).

Header contains technical details about the executable. The most important parts are:

- DOS header — presented for backward compatibility with older versions of Windows
- PE header — executable info such as number of sections, file signature
- Optional header — additional info such as the base of data, base of code, address of entry point
- Sections table — definition of the loading process
- Import/export address table (*IAT*) — lookup table (pointers) for imported or exported structures

Sections contain mainly the data and code of the executable file. The most important parts are:

- Code — actual program code (*.text*)
- Imports/exports — *.idata, .edata*
- Data — static constants (*.data*), variables(*.bss*), other resources (*.rsrc*), debug data (*.rdata*)

The loading process of a PE file starts by parsing and validity checking of the headers and sections table. The file is mapped into memory according to the information in specific header parts. Additional DLLs are loaded into memory based on *IAT*. Finally, the file is executed at a specified entry point. [1]

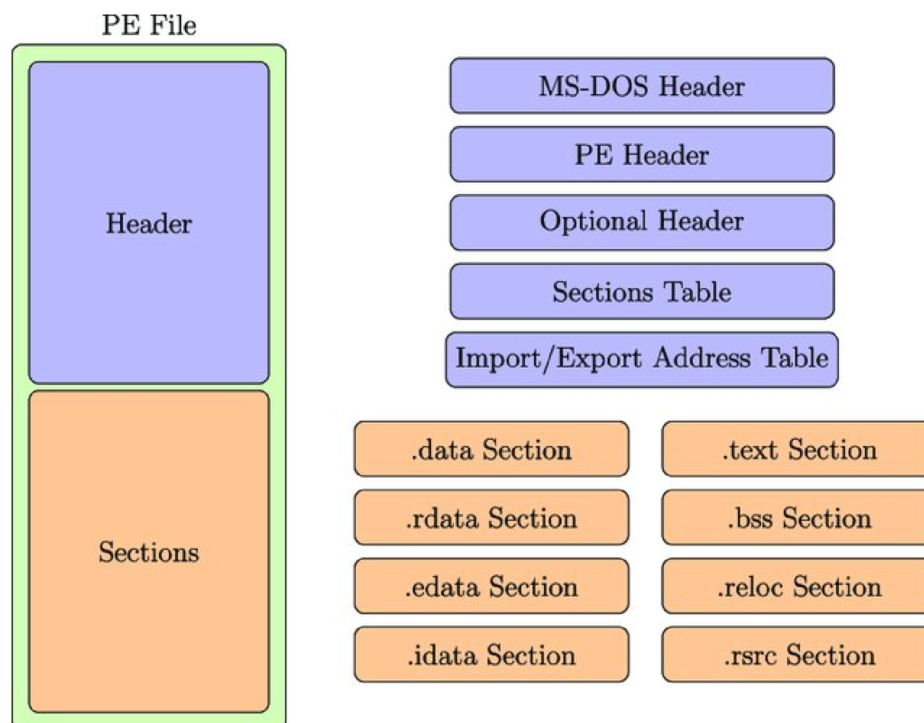


Figure 2.1: PE file structure [39]

2.3 Malware analysis

Malware analysis is a process leading to a deeper understanding of malware features, behaviour, and goals. We can use several techniques to analyze a malware sample, and not all of them require its execution.

Malware analysis is the most relevant if we do not have the source code of the malware binary because its examination would often give us much more information. We see the sample as a black box [93].

[1] https://github.com/corkami/pics/blob/master/binary/pe101/pe101.png

The goal of the security threats research and description is to find a proper response — prevent or detect future attacks, identify the attacker, etc. Authors of [53] summarize our motives in malware analysis as follows. We want to find malware components, their role, and address their goals. For example, we can identify malware consisting of the dropper part, which checks the running environment and downloading payload from the internet, and from the launcher part, which is responsible for the payload stealth execution.

Another reason might be to understand the malware's impact. It is hiding each step, and if we do not monitor our system (for example, using integrity checks), we might not realize the threat. We want to observe and report its behaviour in the target computer (file names, registries, etc.). We also need to detect network intrusion, which might be critical for preventing the malware from spreading across a local and global network.

Finally, we want to classify the threat and investigate who the attackers are and what are their motives.

The analysis might be followed by a response, which often includes an antivirus update — creating new signatures, updating existing ones, etc. The notion of *signature* refers to an indicator that can detect malicious code on victim machines or even in the network traffic [93]. The signature detection is based on examining the outputs of malware analysis and checking specific parts.

We distinguish two basic types of malware analysis — *static* and *dynamic*. In the following sections, we address each of these and describe their techniques.

2.3.1 Static analysis

By performing static analysis, we can examine the target file without its execution. Its usage is limited, but it can narrow the scope of our interest [53]. Performing the static analysis is usually easier and faster [93].

Determining file type

Earlier in this chapter, we listed several file types frequently used by malware. One of the initial steps in static analysis is file type determination. The file type might correlate with the file extension, e.g., *.exe, .sys, .docx*, but reliable technique of its determination is file signature examination. This signature is a sequence of bytes that is unique across different file types. File signature might be examined manually in some *hex editor*[2] or automatically using some programming language libraries or tools[3] [53].

Fingerprinting, comparison

Fingerprinting means generating a cryptographic hash value for the examined file, e.g., *MD5, SHA1*. This hash should give us a unique file identifier that might be used to retrieve known information about the malware sample from online sources[4], where we can find

[2]https://mh-nexus.de/en/hxd/
[3]https://man7.org/linux/man-pages/man1/file.1.html
[4]https://www.virustotal.com/gui/

multi-antivirus scans and other helpful information. On the other hand, we can use a special kind of 'hashes' to identify similar malware samples. Those could be fuzzy hashes, import hashes, or section hashes [53].

String extraction

Strings are placed in the PE file encoded by *ASCII* or another encoding. Their extraction from the original binary is a valuable tool of static analysis. The strings might include IP addresses, domain names, file names and others. There are specialized tools for this task [5].

However, attackers know all these detection techniques, including string extraction. That leads us to the file *obfuscation*, a technique used by an attacker to secure strings from extraction, which is often done by *packers* using compression or by *cryptors* using encryption. Even such techniques are detectable and vincible using specialized tools.

Code analysis

The techniques listed above are picking specific features, and by connecting them, we might get a valuable summary for an initial hypothesis. However, even without the knowledge of the code, we can examine its low-level steps. That is called code analysis, and we can distinguish *static* and *dynamic* code analysis (CA).

In *static* CA we use *disassembler* [6] which translates machine code back to assembly code which might be further analysed. Another option might be *decompilation* which translates machine code into a higher-level programming language such as C or Python [7].

In *dynamic* CA, we use *debugger* to examine the translated code during the run [53] (this technique is part of dynamic analysis, we mention it here for completeness).

There are even other techniques, e.g., *PE header inspection* where we can see imported and exported libraries and functions [93] or *Yara rules*, which allow researchers to create indication rules based on the textual and binary information of the malware sample [53].

2.3.2 Dynamic analysis

Dynamic analyses techniques allow us to examine a running malware sample. The isolated environment for malware execution is often called *Sandbox*. Sometimes we call *Sandbox* the application, which allows us to orchestrate the dynamic analysis and all its parts. During the run, we observe the details about the malware behaviour and even the system's reaction. In the following list, we can see different subjects which are often monitored.

- Processes — process activity, subprocesses, etc.
- File system — dropped files, removed files, etc.

[5]http://split-code.com/strings2.html
[6]https://binary.ninja/
[7]https://github.com/avast/retdec

- Registry keys — read/write operations with Windows registry keys, including read and written data
- Network activity monitoring — outgoing and incoming traffic
- API calls — external library of an operating system which is called by the malware sample, essentially everything what the malware performs should be expressed in the list of API calls
- Mutexes — flags which are usually created by a thread to avoid another thread from writing to a specific resource at the same time, they are also used by malware for interprocess communication, e.g., the indication of the malware presence on the victim machine

Sandbox realization is precise work. We need to minimize the risk of malware breaking the border of a safe environment. It is usually implemented using a virtual machine, and *air-gapped* networks (isolated networks) [93]. Using virtual machines is better for overall security, but there are some significant pitfalls as well. The crucial drawback is that malware might identify the suspiciously clean and safe environment and shut down itself before any action. The sandbox setup has to be conscientious to imitate the natural environment faithfully. We should also let some intentional traces of everyday usage as mentioned in [15]. Despite mentioned facts, virtual machine setup is still more frequent than running malware on a physical machine. There is various virtualization software[8] that provides virtual machine management tools. Using a sophisticated network setup (bridged network adapters, NAT, VPN), we may provide even a controlled internet connection (or simulation).

Examples of existing solutions for sandbox environment orchestration are:

- CAPEv2 — https://www.capesandbox.com/ (earlier cuckoo sandbox)
- ANY.RUN — https://app.any.run/
- Hybrid analysis — https://www.hybrid-analysis.com/
- Joe Sandbox — https://www.joesandbox.com/

Sandbox evasion techniques

As we mentioned earlier, the creators of malware know how it is examined, and they might utilize evasion techniques to detect the sandbox. Malware might delay its execution to overcome the timeout of most sandboxes (usually up to half an hour). It often checks the unlikely storage size, version, and other information overlooked during the virtual machine setup. It might detect a low number of CPU cores and other environment details before dropping a payload. If the sandbox runs on a device with GUI, malware can try user interaction detection [87]. A comprehensive description of sandbox evasion techniques can be found in [3] and this blog post [19] describes how we can defeat them.

[8] https://www.virtualbox.org/, https://www.linux-kvm.org/

2.4 CAPEv2

An open-source project called *Cuckoo sandbox* was firstly published in 2011. It started as the Google Summer of Code project in 2010 within the Honeynet Project. This project is no more actively developed but in 2019, community developers forked the original project and updated its implementation to be compatible with python 3[9]. This project is called *CAPEv2*, and it is distributed under GPL-3.0 License. There is one publication about original *Cuckoo* sandbox [74], which is a partial source of the following description together with documentation of *CAPEv2* [15]. Other sources are referred to in the text.

CAPEv2 is used to automatically run malicious files, collect results, and perform further analysis. It has a modular design which allows its integration into a more complex infrastructure. Other developers can customize and extend many of its functions.

It can analyze various file types, which can be uploaded using CLI or the web interface. We can see the results in the file system or the web application [10]. A list of file types that can be analyzed using *CAPEv2* can be found in B.

2.4.1 Architecture

CAPE's architecture is demonstrated in figure 2.2. It consists of one or more *host* devices. Each *host* might manage multiple *guests* — virtual machines.

Host is the environment for sandbox application usually running Linux distribution[11]. It allows us to upload a new sample, retrieve results, configure the sandbox, and many other functions.

Guests are virtual machines where particular samples usually run under Windows 7 OS. By default, guests are in the isolated virtual network where they can not access each other, only the *host*.

2.4.2 Components

The sandbox consists internally of several components. They can be seen in figure 2.3 and their description follows.

Scheduler

This component runs on *host* machine. It checks configured limits and proper setup. It manages machinery modules and starts a new task if there is a pending *guest*. A new task is forwarded to the *analysis manager*.

Analysis manager

Analysis manager is responsible for the whole flow of a single task. It starts other modules which are part of the analysis and starts/stops the machine through *machinery modules*.

[9] https://github.com/kevoreilly/CAPEv2
[10] public instance https://capesandbox.com/
[11] recommended https://releases.ubuntu.com/20.04/

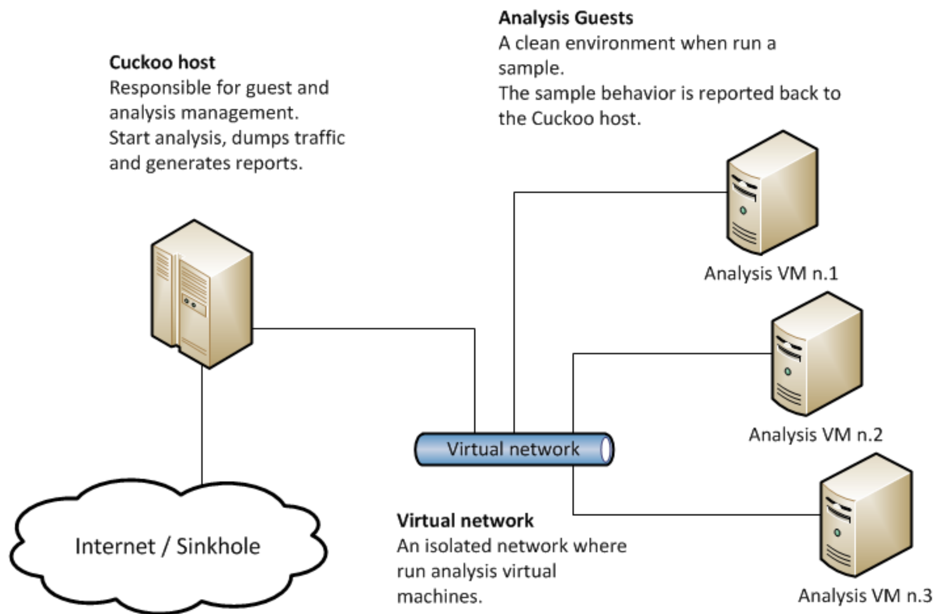


Figure 2.2: Official image of sandbox architecture [15]

Machinary modules

These modules are used by sandbox to manage virtual machines — start, stop, restore. They are initialized during sandbox startup. Recommended software for virtual machine management in *CAPEv2* is *KVM*.

Guest manager

This component communicates with the *agent*. It uploads the sample, checks the state of the analysis and the machine, shutdowns the machine in case of timeout.

Cape agent

Agent is an HTTP server running inside the *guest* machine to report its state and allow sample upload and related actions. It has to start simultaneously with the machine startup (it has to be added to the Windows Startup directory).

Analyzer

It is a platform-dependent software running in *guest* machine that controls the flow in the machine. It is started by *agent* and its configuration is provided per analysis. It runs the sample using a chosen package (specific for each type of uploaded file). If the package is not provided, it might be determined automatically. After running the target sample, *analyzer* injects it with the *cape monitor*.

Cape monitor

A DLL which is injected into the running sample. It logs any captured behaviour using several techniques like hooking functions, following processes, and PE dumping. It also sends results to the *result server*. *Cape monitor* is maintained in separate repository[12].

Auxiliary modules

These are additional modules that run before or during the analysis in the *guest* machine. An example of such a module is the network traffic sniffer or the screenshot capture tool.

Result server

It collects the analysis results and stores them. It runs on the *host* machine.

Processing modules

Processing consists of the following parts - raw data processing, signature matching, and reporting. The first part transforms the raw output to a readable/searchable format, performs static analysis, and extracts network streams. There is a structured output at the end. The second part is running particular signatures and collecting their results. Signatures are stored in a special repository[13]. Their interface accepts the processed structured data and generates results indicating that the current signature matches (true or false) and related data. Signature results are added to the structured output. The final part of the analysis is reporting. In this part, all results are stored in *JSON* report and also in the database (other reporting modules might be added).

The list of raw data processing modules:

- AnalysisInfo — parses basic information about analysis
- BehaviorAnalysis — parses the raw behavioural logs and performs some initial transformations and interpretations, including the complete processes tracing, behavioural summary and process tree
- Debug — parses errors and analysis.log generated by the analyzer
- Dropped — parses information on the dropped files by the malware and dumped by *CAPE*
- Memory — executes Volatility analysis on a full memory dump
- NetworkAnalysis — parses the PCAP file and extracts some network information, such as DNS traffic, domains, IPs, HTTP requests, IRC and SMTP traffic
- ProcMemory — performs analysis of process memory dump
- StaticAnalysis — performs static analysis of PE files

[12]https://github.com/kevoreilly/capemon
[13]https://github.com/kevoreilly/community

- Strings — extracts strings from the analyzed binary

A Signature might isolate some unique behaviour, identify some malware family or type, and spot significant modifications performed in the system. The signature report consists of an identifier, description, severity, category, and other related information.

2.4.3 Analysis flow

A target file upload might be performed using the CLI utility of *CAPEv2*, using the Web interface, Python API, or REST API.

If a file is uploaded, *CAPEv2* saves it to the database. Multiple options can be configured for each analysis — used package, machine to run on, network setup, timeout, priority, and multiple additional options. We can run only network analysis or only static analysis. The *scheduler* keeps track of pending tasks and runs them. The process of execution is managed by *analysis manager*. In case of running a new analysis, *analysis manager* informs the *result server*. When the analysis is running, the *analyzer, monitor*, configuration and the sample file are uploaded to the *guest* using the *agent*. The *agent* starts the *analyzer*, run the sample and injects the *monitor* to that. The *analyzer* and the *monitor* sends results to the *result server*. After the analysis stops or timeout passes, the *analysis manager* stops the machine. The collected analysis results are forwarded to the *processing modules*. Results are stored in the chosen formats and saved to the database [99]. The whole flow can be seen in figure 2.3.

2.4.4 Network setup options

CAPEv2 provides multiple possibilities for the *guest* network setup which could be configured per analysis.

In the case of *None* routing, the machine is isolated, and the only connection is the one with the result server. Additionally, there is *Drop* routing when all the traffic is actively dropped (a more aggressive option).

Other options provide internet connection (in some sense). *Internet* routing is full internet access through a specified interface. We can also forward the traffic through another gateway, i.e. *VPN*, *SOCKS5* proxy[14] or *Tor*[15]. Last option is to use network simulation like *InetSim*[16].

2.4.5 Other features

One of the crucial features of *CAPEv2* is the debugger, so we can also perform non-interactive dynamic code analysis. It allows dynamic anti-evasion bypasses such as the one used by *Guloader, Ursnif* or *Zloader*.

[14] example tool https://github.com/RicoVZ/socks5man

[15] https://www.torproject.org/

[16] https://www.inetsim.org/

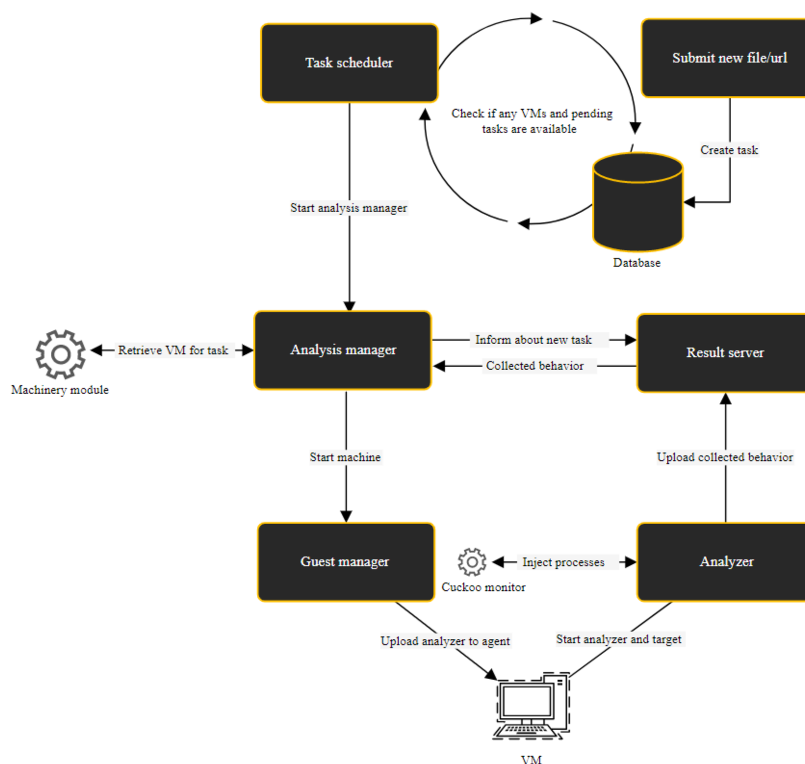


Figure 2.3: *CAPEv2* components and analysis flow [99]

CAPE means *Config And Payload Extraction*. The main motivation behind its creation was malware payload extraction, for which it uses techniques like Process injection, Decompression of executable modules in memory or extraction of executable modules. *CAPEv2* automatically creates a process dump for each process that effectively detects the basic packers.

It can detect various malware families such as Emotet, QakBot, Dridex, and many others. It also uses Yara rules.

2.4.6 Produced data

The whole ouput of *CAPEv2* is listed in the table 2.1.

Output

2.4.6.1 Report

As we stated in the thesis introduction, our goal is to use *behavioural* features and *signatures* from the *JSON* report, which is the most comprehensive report produced by *CAPEv2*. We

Table 2.1: CAPEv2 Sandbox output

Output	**Description**
pcap report	network traffic record (packet sequences)
memory dump	dump of RAM (its analysis results could be also presented)
bingraph	mechanism that discovers metamorphic malware [61]
behavioral log	raw logs of api calls and other (usually in *BSON* format)
dropped files	all dropped files unchanged in separate directory
CAPE, procdump	other extracted payload in separate directory
reports	JSON report and possibly other reports
screenshots	taken during analysis

will use its format as a direct input for our model. Let us define this format [17] and the content of the report.

JSON (JavaScript Object Notation) is the most frequently used lightweight data interchange format. It consists of two essential structures — *collections of key-value pairs* (sometimes called *objects*) and *ordered lists*.

Object is an unordered list of *key-value* pairs, curly brackets surround it, pairs are coma-separated, and a colon separates keys from values. *Keys* have to be double-quoted Unicode strings. *Values* might be strings, numbers, objects, arrays, boolean or null. *Lists* are surrounded by square brackets and contain comma-separated values.

Usually, a single *.json* file contains one object, but there are also cases where a list of objects is presented.

The *CAPEv2* report usually has tens of megabytes but sometimes even gigabytes. Complete schema of the report is in B.1. In the thesis attachment, we can see an example of a real report (H). In the modelling part, we will concentrate on *signatures* and the *behaviour* parts, its structure is described in B.2.

2.5 Prior arts

Below we list relevant publications where the authors applied a machine learning algorithm to the data produced by malware analysis tools.

[17] documentation https://www.json.org/json-en.html

Static features

- Strings — [64]
- N-grams (analysis of byte subsequences of length N from the original binary) — [34]
- Entropy of a malware file — [101]
- Statically extracted API function calls — [4]

Dynamic features

- Registry — [38]
- CPU instruction traces — [17]
- Network traffic — [14]
- API call traces — [35]

Other related resources might be [95, 91, 1, 39].

Chapter 3

Malware classification

In this chapter, we describe the machine learning background, which is important for further model description. We cover basic terms in machine learning, cybersecurity context, and models for hierarchically structured data (*JSON* documents).

3.1 Machine learning

"Learning is a search through the space of possible hypotheses for one that will perform well, even on new examples beyond the training set. To measure the accuracy of a hypothesis, we give it a test set of distinct examples from the training set." [89]

As we can see in the quote, the essentials of machine learning can be described concisely and elegantly. On the other hand, the crux is its mathematical base. The most considerable challenge in machine learning theory is the formal framework and point of view, which the authors consider. Firstly, we would like to put our method into context.

After a brief look into several sources, we can find varied kinds of machine learning or just learning [89]. It is easy to mix perspectives and make one extensive taxonomy. However, we start with statistical machine learning as the most general notion in this chapter.

In statistical machine learning, we aim to optimize the predictive function to fit our training data and perform sufficiently on testing data. It is done using statistical tools such as *maximum likelihood estimation* and many others. On the opposite, we can see symbolic learning where we are more interested in symbolic knowledge representation, often human-readable. This approach is older and sometimes called *GOFAI* — Good Old-Fashioned Artificial Intelligence [43]. This kind of learning is not going through such a massive upswing as the statistical branch nowadays. Let us define the basic terms based on [31].

3.1.1 Definitions

Sample — independent variable set

By sample, we mean a collection of features which tends to be represented as $x \in \mathbb{R}^n$, where x_i is often called feature and x is called sample or feature vector [40]. We can also generalize this definition for tensors.

Generally, we can see object features as $x \in \mathcal{X}$, where $\mathcal{X}$ could be a set of categorical variables, scalars, real-valued vectors, sequences, images, graphs, structured formats (*JSON*) and much more. We might involve a feature extraction process to get to the real-valued vectors mentioned above.

States, classes — dependent variable set

By state, we mean the subject of our prediction, often represented as $y \in \mathcal{Y}$, where $\mathcal{Y}$ is often called *state space*. That could be whatever we enumerated by $\mathcal{X}$ (images, documents, vectors...). They also tend to be represented as real-valued or discrete vectors. States are sometimes also called labels or targets. We focus on classification tasks, so we call them classes.

Prediction strategy, hypothesis

A prediction strategy is defined as $h : \mathcal{Y} \rightarrow \mathcal{X}$. The output of prediction strategy we denote as $h(x) = \hat{y}$, where $h \in \mathcal{H}$ (often called hypothesis class). On the contrary, the real state we denote by y.

Example

Assume the usual situation that before learning, we receive a set of examples to learn from. Based on what we receive, we can distinguish between several types of learning. This thesis works with *supervised* case.

1. *Supervised learning* — example denotes pair (x, y), where $x \in \mathcal{X}$ and $y \in \mathcal{Y}$
2. *Unsupervised learning* — example denotes $x \in \mathcal{X}$
3. *Semi-supervised learning* — each example could be one of the possibilities above

We are usually working with the set of examples that we later divide into different subsets, e.g., *training, testing*, and *validation sets*.

The crucial assumption is that X, Y are random variables related by unknown joint p.d.f[1] $p(x, y)$. This assumption makes the whole learning process reasonable because we assume the relationship between the variables. We also assume that we can draw i.i.d.[2] examples from this p.d.f.

Loss function

Loss function denotes the objective of our optimization task during learning, $\ell : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}^+$. Usually, we compute its value for each particular example $\ell(y, h(x))$ and use some aggregation, e.g., mean, to get one value for the whole set of examples.

[1] Probability Density Function
[2] identically independently distributed

Learning

Main consequence of the assumption about randomness of $\mathcal{X}$ and $\mathcal{Y}$ is that $h(x)$ and $\ell(Y, h(X))$ are also random variables. This fact allows us definition of *expected risk* (3.1).

$$R(h) = \mathbb{E}_{(x,y)\sim p}\ell(Y, h(X)) = \sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\ell(y, h(x)) \tag{3.1}$$

$$h^*(x) = \operatorname*{argmin}_{y'\in\mathcal{Y}} \sum_{y\in\mathcal{Y}} p(y|x)\ell(y, y') \tag{3.2}$$

If $p(x, y)$ is known, optimal prediction strategy would be denoted by (3.2). In practise, it is unknown and we have to involve an approximation (learning algorithm) to find the best attainable strategy using drawn data.

Assume $\mathcal{T}^m$ is a set of examples for supervised learning. We can distinguish two basic learning approaches — *Discriminative learning*, *Generative learning*.

In *Discriminative learning* we assume $h^* \in \mathcal{H}$ and we approximate *expected risk* with *empirical risk* ((3.3)).

$$R_{\mathcal{T}}(h) = \frac{1}{|\mathcal{T}|} \sum_{(x,y)\in\mathcal{T}} \ell(y, h(x)) \tag{3.3}$$

Optimal strategy is denoted by $h^*_{\mathcal{T}}(x) = \operatorname{argmin}_{h'\in\mathcal{H}} R_{\mathcal{T}(h)}$. Models trained using this approach are, for instance, linear regression, support vector machine, and neural networks.

Generative learning assumes that true p.d.f. $p(x, y)$ is part of some parametrized family of distributions. The task for our algorithm is to localize the point estimate of parameters θ based on $\mathcal{T}$.

In our work, we are using *discriminative* models, namely neural networks.

We distinguish two types of error. *Approximation error* $R(h_{\mathcal{H}}) - R^*$ is caused by the choice of $\mathcal{H}$ (choice of model), $R(h_{\mathcal{H}})$ denotes best attainable risk using only hypotheses from $\mathcal{H}$. *Estimation error* $R(h_m) - R(h_{\mathcal{H}})$ where $R(h_m)$ denotes the risk learned from the training data.

3.1.2 Machine learning tasks

Regression

States $y \in \mathcal{Y}$ are continuous-valued tensor in this case, most often $y \in \mathbb{R}^n$. For example, features might be the outputs of static and dynamic detectors and network traffic records, and the state is a risk score represented as a real value ([51]).

Classification

In this case, $y \in \mathcal{Y}$ is categorical vector, in most cases $y \in \{1, \dots, \mathcal{C}\}$, where $\{1, \dots, \mathcal{C}\}$ is encoding for real world values like *man* and *woman*. If $\mathcal{C}$ is 2, than we call the task *binary classification* and if $\mathcal{C} > 2$ we call it *multiclass classification*. We can classify to not mutually exclusive classes at once, that is called *multi-label classification* or *multiple output model* [72]. Classification could be even more complicated, e.g., we can classify into a class hierarchy [107].

Given x classifier outputs $\hat{y}$ which is an encoded class or probability distribution over classes [40]. An example of such a distribution might be the output of the *softmax* activation function in a neural network. This distribution might be later interpreted and used during evaluation and further analysis. We have to determine predictions of such classifier because we have real values instead of discrete classes. In binary classification, the discretization is often done by setting a *treshold*. If the result is above the specified threshold, it is in a positive class and vice versa.

An example of a classification task could be malware classification using the well-known SVM algorithm [59].

Classification and *regression* are not the only variants. There are others like transcription [40] or anomaly detection [20] and many other problems mentioned in [40] or [107].

Anomaly detection is a frequent problem in cybersecurity, where we are interested in detecting what is not matching the usual pattern. That might be related to an unsupervised learning task where we can use algorithms like Expectation Maximization [24]. For example, in this paper [48], we can see the anomaly detection of network traffic data.

Examples of discriminative classification learning algorithms are — Support vector machine, Decision trees, Logistic regression, and neural nets.

Our intention is in *classification*, we focus on it further.

3.2 Loss functions for classification

The learning process builds on function optimization. Our criterion is the chosen loss function. Every loss function has its interpretation, and we choose it based on our goals (domain-specific). We mention two standard functions which are often involved in classification tasks.

Multinomial logistic loss

We assume that the outputs of our model are conditional probabilites $p(c|x;\theta)$ for each class $c \in \mathcal{C}$ and each of n examples. The model's parameters are denoted by θ. Multiclass cross entropy is defined in (3.4). This loss function is sometimes called *multiclass cross entropy*.

$$\ell(\theta) = -\frac{1}{n} \sum_{i=1}^{n} \sum_{c \in \mathcal{C}} \mathbb{1}\{c == c_i\} \log p(c|x;\theta) \tag{3.4}$$

Its idea is to minimize values representing the average logarithm of the probability of the actual class (denoted by c_i) across examples, correctly predicted by the model.

Hinge loss

This loss was introduced in [37] and its well-known usage we can find in the Support vector machine algorithm. The formula could be seen in the figure (3.5), where $y \in \mathcal{Y} = \{\pm 1\}$ denotes truth label and $\hat{y}$ denotes the classifier's score.

$$\ell(y) = \max(0, 1 - \hat{y} \cdot y) \tag{3.5}$$

3.3 Model Evaluation

Based on our domain and goal, we have to choose proper classifier evaluation metrics. The majority of such techniques do not depend on the type of model we have. The classifier itself is just a black box, and we evaluate its predictions.

Learning is often divided into two or three phases — *training*, *testing* (sometimes *validation*). Metrics are connected to the data, e.g., accuracy on the training set is different from accuracy on the testing set.

Evaluation metrics are most often used to measure the quality of the model. It also might be used to compare the results of different approaches. If we tune the model's hyperparameters, we often monitor accuracy on the validation set.

Loss function

The loss function is the objective with its meaning and possible interpretation (described above). Its value is often monitored directly during the learning process. We might observe the loss function value difference between the training set and the validation set as a possible overfitting indicator.

Confusion matrix

The most significant metrics are derived from the concept of *Confusion matrix* (table 3.1. For our convenience, we define a confusion matrix for a binary classifier. Its generalization for the multiclass case is just larger, but the idea is the same. Common problems in binary classification are formulated in a way that $y \in \{positive, negative\}$. As an example, we can introduce the classification that a patient has cancer or not. All (x, y) where $y = positive$ we call positive examples and the opposite examples are negative.

Table 3.1: Confusion matrix for binary classifier

		Ground truth	
		Positive	Negative
Classified	Positive	True positive (TP)	False positive(FP)
	Negative	False negative (FN)	True negative (TN)

Before deriving basic metrics, we have to emphasize that they have to be treated in a particular context. The most important condition is the overall balance of the dataset — the ratio of positive and negative examples. List of important metrics is in A.1.

Very often, we can see also curves which are plotted along reported metrics. These plots fit in situation when we are trying to compare multiple classifiers. Most seen are *ROC* (described in [29]) and *PRC* (described in [30]). Data points for these two curves are collected by iterating over possible *treshold* values and for each we calculate specific metric. In case of *ROC* metrics are *FPR* on x-axis and *TPR* on y-axis. In case of *PRC* metrics are *TPR* (sometimes *recall*) on x-axis and *precision* on y-axis.

The *PRC* is better in the case of an imbalanced dataset where we have a larger number of negative examples, and we especially care about positive examples and their predictions.

If we need to have a single number as a performance metric, including multiple thresholds, we can use the area under the *ROC* or *PRC*.

Cyber Security context

Some metrics are more critical than others in cybersecurity. It is crucial to think about domain-specific facts choosing appropriate metric to measure our model's performance.

The class imbalance is one of the challenges in cybersecurity. If the dataset consists of 80 % of positive examples, then a classifier predicting only positive class has $accuracy = 0.8$. In [45], we can see the usage of *geometrical mean* to deal with this issue. We can also use *balanced accuracy* to cover the dataset's imbalance in the metric calculation.

During Intrusion Detection in cybersecurity, false negative examples can be a potential security risk for the target subject (person, company, state etc.), so it is often the priority number one. On the other hand, *False positive* classification means false alarm, which costs employee time and trust [75]. The frequency of software creation is significant, so even a relatively low false positive rate can cause the security team to solve something harmless instead of a real risk [8]. In malware detection, that could also be reinforced by the fact that the protection is too aggressive. Such results can lead us to recurring expenses [60]. In such a case, we can use *ROC* curve with *logarithm scale* on the x-axis to observe even a low false positive rate. Note that our dataset has to be sufficiently large to observe a statistically significant estimate of a low positive rate.

To sum up, we have to find an optimal point where the model predicts attacks successfully, but still with a low number of false alarms.

In the thesis, we will work with imbalanced datasets (more negative examples). We use several metrics mentioned above especially *balanced accuracy*, *ROC* curve with logarithm scale on x-axis, and *PR curve*.

3.4 Neural Networks

A neural net is a discriminative model which is based on *Empirical Risk* minimization (3.3). The neural network is a parametric function created by composing simple functions. Specifically, $nn(x) = g_1(g_2(g_3(\ldots(g_n(x, \theta_n), \ldots; \theta_1)$, where each g_n has the form $\sigma(Wx + b)$ and

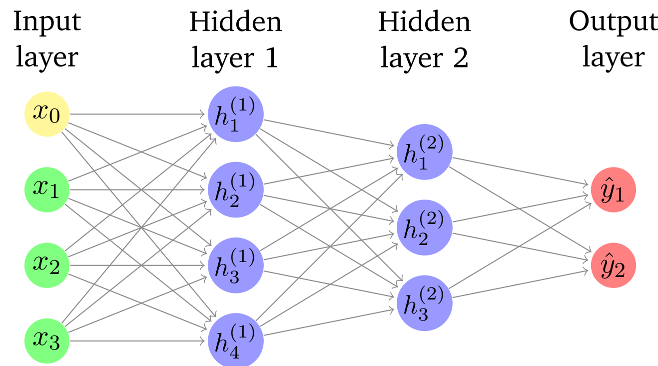


Figure 3.1: Neural net example

θ denotes parameters (W, b). Functions, their input and output are often demonstrated as layers of a neural net. An example of a general neural net could be seen in 3.1. The input layer represents items of the feature vector. A *multilayer perceptron* described in [89] with a nonlinear activation function might realize hidden layers. The output layer represents a predicted state. Many functions are used in neural nets, e.g., before the output layer is often a softmax or sigmoid activation function to normalize the input into a probability distribution (score). The overall goal is to optimize ℓ with respect to the parameters of the net. Usually, we use *gradient descent* optimization technique.

Usage of the *gradient descent* puts no extra demands on the data we are using. We also do not insist on the strict convexity of the function we are optimizing (the function does not have to have one global *minimum*). The assumption about all functions is that they have to be (at least piecewise) differentiable with respect to their inputs and parameters.

Backpropagation algorithm is used for the effective gradient computation [88]. This algorithm allows us computation of the derivative of ℓ with respect to all network parameters. The main idea builds on computing derivatives of every function's output with respect to its input. Then by applying the *chain rule*, we can propagate the information from the following layers to the previous ones.

The optimization is most often done by *stochastic gradient descent* [56]. This variant of the *gradient descent* algorithm is usable even in huge datasets or online learning. Usually, we divide the original training dataset into subsets (we call them *batches* or *minibatches*), and we use them to compute the gradient rather than the whole dataset. That is done for a specified number of iterations. *Batch* is randomly subsampled from the training set, which is the reason for '*stochastic*' in the method name. The *batch* size and the number of iterations are hyperparameters of the neural net.

3.5 Tree-structured data classification

Real-world use cases often provide more complex datasets than just fixed dimension matrices or images. As we mentioned, malware analysis data are stored as *JSON* files, and modelling such data is our goal. Those files could be formally seen as tree-structured in-

puts (more generally graphs). In the following section, we describe two straightforward approaches of data classification and two more sophisticated.

3.5.1 Rules

An example of a rule might be that if we observe a specific value in a specific key part in the document, e.g., {" api_calls ": "DeleteFileA"}, we classify the current document to a specific class, e.g., *dropper*. The logic might be more complex. We can count a score aggregating more information about the current document and classify the document according to the score. As an example of such a rule-based approach, we can see malware signatures mentioned in the previous chapter. Their implementation deterministically checks some part of the original *JSON* report and performs binary classification (*positive* — the signature is added to the report, *negative* — the signature is not added to the report). The rules are usually defined manually.

3.5.2 Flattening

This approach deals with the data structure itself, which has no fixed dimensions. It is more a feature engineering technique than a learning approach. Each document might contain a different number of keys in a different order, and the size of arrays may vary. Using flattening, we find a mapping/procedure which allows vectorization of each document. The target learning algorithm is used on the flattened dataset. An advantage of this approach is that we can use a plethora of off-the-shelf algorithms, but on the other hand, we reduce the hypothesis class.
The motivation of more complex techniques is that the data structure also keeps some information, and it is worth trying to model even the structure, not only the data.

3.5.3 Graph neural networks

Let us firstly define a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes a finite set of vertices and $\mathcal{E} \subseteq \binom{\mathcal{V}}{2}$ for indirected graphs or $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ for directed graph denotes a finite set of edges.

Graph neural network (GNN) was introduced in [90], and it is a suitable model for problems represented by a graph (directed or undirected). Initial setup is that the problem representation is $G(\mathcal{V}, \mathcal{E})$ where each vertex is associated with its embedded value v_i, where $i \in \{1, \ldots, |\mathcal{V}|\}$ denotes index of vertex in graph. By an embedded value, we usually mean representation of vertex information in $\mathbb{R}^m$. The output of a GNN is typically the same structured graph (same edges and vertices) with optimized v_i. The output graph might be used in various ways — calculate the loss for the next iteration, calculate an aggregation of all values, interpretation of particular v_i. The usual task for GNN is node selection, node classification, or graph classification. The most prevalent method for GNN optimization is based on message passing. Each iteration of such GNN consists of three steps:

- Compute a message for each of the chosen pairs (v_i, v_j) of vertices (might be all pairs, just neighbouring or other) using values from the previous iteration
- Aggregate messages for each vertex i by an aggregation function

- Update v_i using aggregated message for $i \in \{1, \ldots, |\mathcal{V}|\}$

Assuming that all functions in the net are at least piecewise differentiable with respect to their parameters, we can use the stochastic gradient descent optimization approach.

The natural structure of *JSON* document is a tree which is a less general example of a graph. Thus, in theory, GNNs can be used to classify *JSON* documents, yet the approach might be unnecessary computationally expensive [78].

3.5.4 HMill framework

In our experiments, we aim specifically at the hierarchical multiple instance learning (*HMill*) framework defined in [68] because it was designed specifically for tree-structured data and it uses the structure of the input for the model optimization. The authors proposed a use case of *JSON* modelling with good results, and we would like to demonstrate the framework's performance on more complex data. In comparison to the GNNs approach, *HMill* model has better scalability, and it is computationally efficient since a single pass over the data is sufficient (unlike in GNN where you need multiple passes) [68]. Authors adopted and proved the universal approximation theorem [46] in the *HMill* situation which shows that *HMill* can approximate any continuous (measurable) function from the space of *JSON* documents to R^n. Concerning mentioned facts, we believe *HMill* is more suited for our problem. We will describe this framework in the next chapter.

Another example of structured data modelling motivated by recurrent and recursive neural networks is in [102]. More generally, graph-structured data modelling is part of [44] or [13].

Chapter 4

Hierarchical Multiple Instance learning

The previous chapter summarized machine learning formalism and ended with structured data modelling. We want to model specific parts of *JSON* report produced by the *CAPEv2* sandbox, as we stated in the introduction. There is a vast amount of information in the behavioural part of the report. Sometimes we can find the same values in different parts of *JSON* document, which shows that the document's structure also keeps some semantics that might contribute to the prediction. This chapter describes multiple instance learning and the *HMill* framework. At the very end, we describe our modelling case in detail.

4.1 Multiple instance learning

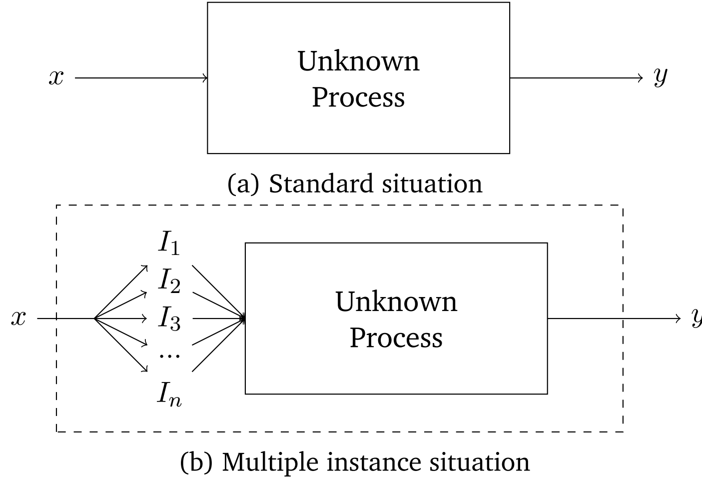
At first, let us describe and define the problem of *Multiple instance learning* (*Mill*), its origin, and formalism. This term firstly appeared in [25]. However, it is not the very first formulation of such a problem, that is in [54].

The motivation example in the original paper [25] is formulated in the following way. Imagine we have multiple keys and one door, and some of the keys unlock the door, and some do not. Note that we cannot access the door, we only get the keys with labels. In standard learning, our goal is to learn a classification model, which consumes a key and outputs if it can open the door. However, in *multiple instance learning* setting, we receive whole key chains with various keys. Each chain is assigned with a label showing if it contains a key opening the door or not. Our goal is to learn a classification model, which consumes a key chain and outputs if it can open the door.

In the figure 4.1 we can see the basic idea behind *Mill* problem (I_i denote instance). The only significant difference regarding the previous chapter is the definition of an *Example*.

Example

We assume a *supervised* learning setting. Our examples consist of two parts — *bag* b and *label* y. A label was defined earlier and its meaning is the same, but it is related to a bag and



(a) Standard situation

(b) Multiple instance situation

Figure 4.1: Supervised learning

sometimes called *bag label*. *Bag* b is a set of feature vectors $x_i \in \mathcal{X}$, these vectors are called *instances* and usually lives in the standard feature space $\mathbb{R}^d$. The cardinality of each bag is $|b| \in \mathbb{N}$ and can be even zero. We assume $b \in \mathcal{B}$ which denotes a *bag space*. *Bag space* might also be seen as $\mathcal{B} = 2^{\mathcal{X}}$ which denotes all finite multisets of $\mathcal{X}$ (repetition of item within a set is allowed). We can see a standard learning situation described in chapter 3 corresponds to a *Mill* problem where holds $|b| = 1$.

For the thesis we assume *Mill* classification problem so $|\mathcal{Y}|$ is finite (typically binary classification $\mathcal{Y} \in \{positive, negative\}$).

4.2 Mill problem solution

The goal of the multiple instance learning process is identical to that in the standard setting. This section describes three general approaches (paradigms) of solving a *Mill* problem. They are formulated in [6] as *Instance-Space, Bag-Space* and *Embedded-Space* paradigm.

4.2.1 Instance-Space paradigm

An algorithm in this group infers an *instance-based* classifier $f(x) \in \{1, 0\}$ that classifies each instance x in the training set. *Bag-based* classifier is constructed in the way shown in (4.1), where x_i are instances in the bag b, $\circ$ denotes an algorithm-specific aggregation operator and Z denotes an optional normalization factor. As we stated earlier, *bag* labels are part of the training set but *instance labels* are not. We have to make some assumptions about the relation between *bag* and *instance* labels, if we want to use the method building on Instance-space paradigm.

$$F(b) = \frac{f(x_1) \circ f(x_2) \circ \cdots \circ f(x_N)}{Z} \tag{4.1}$$

4.2.1.1 Standard assumption

We assume that each negative bag consists of negative instances only, and each positive bag includes at least one positive instance. The algorithm in this setup aims at instances that make bags positive (we know that at least one in each bag does that).
There are several methods that follow this assumption. The first is *Axis-Parallel Rectangle* used in [25] in drug discovery use case, where $F(b) = \max_{x \in b} f(x)$. Other methods are *Diverse Density* [69] or MI-SVM [7].

4.2.1.2 Collective assumption

Previous methods tend to look over the fact that the bag label might be influenced by the interaction of features from different instances. Some of them might consider only several instances (or even one) from the whole bag.
Collective assumption states that “all instances in a bag contribute equally to the bag’s label” [103]. Methods usually use a training set of instances which is constructed such that each instance inherits its bag’s label.
This training set might be used to get an instance-level classifier. A basic approach is to use the SIL algorithm [16], which train the mentioned instance-level classifier using SVM, and the bag-level classifier is obtained by (4.2). Another method is Wrapper MI [32].

$$F(b) = \frac{1}{|b|} \sum_{x \in b} f(x) \tag{4.2}$$

4.2.2 Bag-Space paradigm

In this setup, we treat the whole bag to learn the classifier. The discriminant learning process is performed in *bag space* directly in contrast to the previous paradigm where we assumed instance-level classifiers.
Bag space is non-vectorial, but we are able to define a distance function $D(b_1, b_2)$, where b_1, b_2 are bags and the result is representing a measure of their similarity (or distance). Then we can use standard distance-based classifiers such as K-NN.
Assume instances live in *d-dimensional* space such as $\mathbb{R}^d$. We can see them as points, so a bag is a set of points. The problem of computing the distance between two sets of points is well studied. An example of distance function is the minimal Hausdorff distance (4.3), which denotes the distance between the closest points of bags b_1 and b_2 [100]. We can also use kernel functions $K(b_1, b_2)$, which provide similarity measure between bags. An example of such kernel might be (4.4), where $k(x, y)$ denotes instance-level kernel (linear, polynomial etc.) and p is related to the size of the largest bag (in practise found by cross-validation) [36].

$$D(b_1, b_2) = \min_{x_1 \in b_1, x_2 \in b_2} ||x_1 - x_2|| \tag{4.3}$$

$$K(b_1, b_2) = \sum_{x_1 \in b_1, x_2 \in b_2} k(x_1 - x_2)^p \tag{4.4}$$

4.2.3 Embedded-Space paradigm

In the Bag-Space paradigm, the goal was to extract global information from bags. That is achieved by defining a distance function allowing implicit bag comparison. In the Embedded-Space paradigm, we extract the information by defining explicit mapping $\mathcal{M} : b \mapsto v$ from the bag space to a custom feature space which summarizes bag characteristics, where $\mathcal{M}$ is called *embedding*. Based on the choice of $\mathcal{M}$, we may distinguish two approaches — *without vocabulary* and *vocabulary-based* methods.

Without vocabulary approaches make no differentiation among instances in a bag and aggregate overall statistics from each instance of the bag. An example of such an algorithm might be *Simple MI* where the feature vector for a bag is attained by averaging over all instances in it: $\mathcal{M}(b) = \frac{1}{|b|} \sum_{x \in b} x$ [26].

4.2.3.1 Vocabulary-based methods

These methods are in the embedded-space category, and their main idea is to find an embedding based on an instance-level classification. However, instance labels are assumed in a different sense than in the instance-space paradigm. We often involve an unsupervised way to derive the instance-level classifier, so the semantics of an instance label is missing here. Bag embedding is then determined according to the instance labels.

There are three usual components of a *vocalbulary-based method* [6]. *Vocabulary* $\mathcal{V}$ storing instance-level labels (sometimes rather called concepts). Each concept is defined by an identifier and a set of parameters. These concepts are most often created from clusters of K-means algorithm. Second component is a *mapping function* $\mathcal{M}(b, \mathcal{V}) = v$ (embedding) which maps from the *bag space* to a k-dimensional *feature space*, where k denotes the number of concepts. The final component of a *vocabulary-based* method is a standard supervised classifier $G(v) in \{1, 0\}$ which classifies feature vectors in embedded space. Final bag-level classifier results in $F(b) = G(\mathcal{M}(b, \mathcal{V}))$.

We may distinguish several approaches for *vocabulary-based* methods. As an example, we describe *histogram-based* methods. Remaining approaches are *distance-based* and *attribute-based*, for more info we refer to [6].

4.2.3.2 Histogram-based methods

$\mathcal{V}$ denotes resulting clusters of chosen clustering algorithm which outputs K classes $C_1, \ldots C_K$. $\mathcal{M}$ denotes a histogram of classes for instances in a particular bag: $\mathcal{M}(b, \mathcal{V}) = (v_1, v_2 \ldots, v_K)$, where $v_j = \frac{1}{Z} \sum_{x \in b} f_j(x), j = 1, \ldots, K$, where $f_j(x) \in \{1, 0\}$ is likelihood that the instance x belongs to the class C_j and Z is a normalization factor. An example of such an algorithm is Bag-of-Words [73].

A generalization of all listed paradigms is provided in [68]. The authors stated that all approaches require three essential components — function operation at the instance level f, e. g., class classifier, a form of aggregation or pooling g and bag-level classifier F. That leads us to the last method we want to describe.

All three functions are composed together to retrieve a prediction. The idea is to optimize everything together. If all functions f,g and F are (at least piecewise) differentiable with respect to their parameters, we can use gradient descent optimization (neural net). [83, 27] This approach is very flexible as the whole composition is optimized together in contrast to the previous cases where they are designed separately and later connected. Furthermore, no instance labels are required because they are treated implicitly in an unsupervised manner. An example of such a neural network architecture is demonstrated in 4.2. Note that there is one f with shared parameters Θ_f over all instances, the pooling function g might be mean or maximum and final layer F after the pooling gets a fixed dimension bag representation as input.

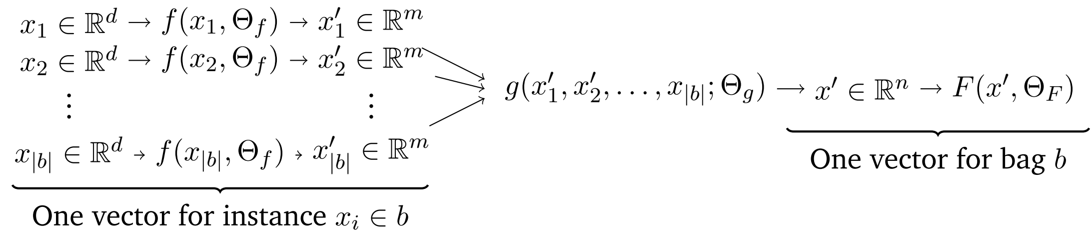


Figure 4.2: Hierarchical *Mill* model by [83] (image inspired by [68])

Demonstration of this approach and its results can be found in [83, 84, 79, 68, 52, 80]. The same idea was published independently as *Deep sets* in [105].

In [68] we can see an application of this approach in the introduced *HMill* framework. Description of this framework follows.

4.3 HMill framework

The neural net architecture defined in [83] was followed by [68] which led to the formulation of *HMill framework* referring to the *Hierarchical Multiple Instance Learning framework*. This framework provides general tools for tree-structured data modelling. Both sample and model consist of various type nodes which are structured as a rooted directed tree. The crucial publication for *HMill framework* is [68] where authors firstly formulated it and demonstrated its use. In this section, we summarize the essential facts. Should the reader wish to learn more, we recommend the original publication [68].

4.3.1 Abstract data nodes

Firstly, let us summarize the basic data nodes in *HMill* framework. The data nodes are used to represent samples. Demonstration of their usage can be seen in figure 4.3.

4.3.1.1 Array node

Our data consist of some low level observations, e.g., strings, booleans, vectors, enums, etc. Authors call these raw observations — *fragments* and $\mathcal{F}$ denotes the space where they are

living. Examples of $\mathcal{F}$ might be Euclidean space $\mathbb{R}^k$, linear space $\mathbb{Z}^2$ or even all strings over finite alphabet. The only requirement for $\mathcal{F}$ is that there must be a mapping $\mathcal{F} \xrightarrow{h} \mathbb{R}^n$. The *array node* an is responsible for storing *fragments* $\alpha \in \mathcal{F}$ in following form: $an(\alpha, \mathcal{F}, h)$. Sometimes the notation is $an(x)$, where $x = h(\alpha)$ means that the transformation was already made.

4.3.1.2 Bag node

Bag node denotes analogy to the *bag* in multipliple instance learning $bn(b)$, where $b = \{a_1, \ldots, a_{|b|}\}$. From the structural point of view, a bag node can be created from a graph node that has:

1. multiple children, which are an with matching $\mathcal{F}$
2. multiple subtrees bn_i with matching structure[1] (nested)

As the instances have to be of the same structure, they form an unordered set in a bag node.

4.3.1.3 Product node

Final node type of *HMill* tree representation of a sample is a *product node*. That is defined as $pn(a_1, \ldots, a_l)$, where $l \geq 1$ and a_i is representation of some *HMill* tree/subtree — *an,bn* or *ps*. The order of a_i is arbitrary but fixed because each abstract node in it might have different dimensions.

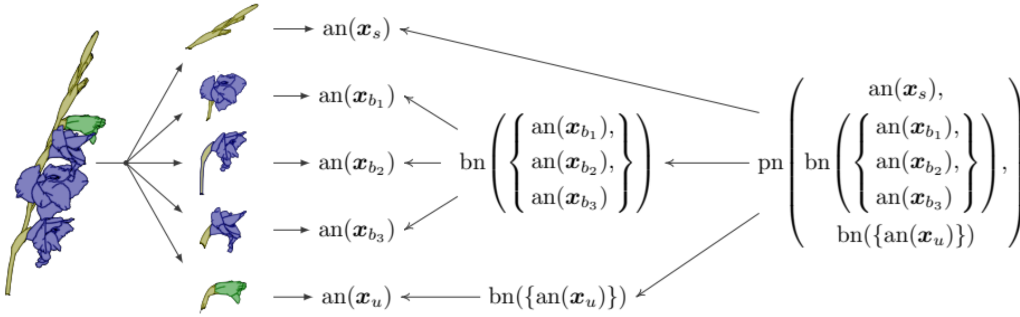


Figure 4.3: Representation of a plant specimen in *HMill* framework [68]

4.3.2 HMill schema

In the *bag node* definition we mentioned that instances in it have to follow the same structure. That is why we define *HMill schema*. Schemas are trees which are mirroring sample trees and for each abstract data node there is a corresponding schema node in the schema structure.

[1]follows the same schema which is defined later

- *Array schema node* ($as(\mathcal{F}', h')$) — defines fragments space instance $\mathcal{F}'$ and mapping h'
- *Bag schema node* ($bs(s')$) — defines subschema (subtree) of instances s' specifying structure of instance in bag node
- *Product schema node* ($ps(s_1, \dots, s_l)$) — defines one or more subschemata s_i of subtrees in the product node

Definition (Schema matching according to [68]). We say that a sample tree t follows a schema s, or matches a schema s, provided the following conditions hold:

- If $t = an(\alpha, \mathcal{F}, h)$ for any $\alpha \in \mathcal{F}$, then t is matching s if and only if $s = as(\mathcal{F}, h)$
- If $t = bn(\{a_1, \dots, a_k\})$, then t is matching s if and only if $s = bs(s')$ and $\forall i \in \{1, \dots, k\} : t_i$ matches s'
- If $t = pn(t_1, \dots, t_l)$, then t is matching s if and only if $s = ps(s_1, \dots, s_k)$,$l = k$ and $\forall i \in \{1, \dots, l\} : t_i$ matches s_i

Note that an empty bag node matches every schema with a bag node in root. If we observe two samples following the same structure and one of them has instances in a specific bag and the second does not, they still might match the same schema.

4.3.3 HMill model

After we defined the nodes and schema, we can take a training set, derive a schema matching all examples, and create a *HMill* samples ready for learning.

Following the idea of the tree structure mirroring in the case of schema definition, we define *HMill* model in the same manner. The goal is to create a model based on the schema extracted from our data, which would accept each sample matching the schema on input. Models consist of model nodes. The model nodes are hierarchically nested functions, each of which outputs one vector given one abstract data node. If we want to perform a prediction for a particular sample, we evaluate the functions in subtrees first and then parents. The root provides the model's output. All nodes are piecewise differentiable with respect to their parameters, which allows us to adopt training methods known from feedforward/convolutional neural networks without any change.

4.3.3.1 Array model — $am(f)$

This model transforms the leaves of the sample tree — fragments. We assume that the mapping h is already applied for the original fragment, so they are in $\mathbb{R}^m$. This model is denoted by $f : \mathbb{R}^m \to \mathbb{R}^n$. We might use any piecewise differentiable mapping, but the authors refer to dense feedforward neural networks.

4.3.3.2 Bag model — $bm(f, g, F)$

A Bag model is motivated by the solution of a multiple instance learning problem provided in [83]. Each bag model can be seen as a multiple instance learning solver for the problem defined by the corresponding bag node and its instances. There are three composed functions — f denotes *instance model*, g denotes *aggregation* and F denotes *bag mapping*. We apply f for each instance/subtree a_i in $bn(a_1, \ldots, a_k)$. All results are input for one or more aggregation functions g, and its output is a single vector. Finally, the *bag mapping* F is applied. The output lives in $\mathbb{R}^d$, where d is the output dimension of a particular multiple instance model. If the bag model is in the root of the tree, d is the overall output dimension of the model. We often use $d = |C|$ as a number of classes. The output is then interpreted as logits of the probabilities of classes.

f might be an arbitrarily complex HMill model. Note that all nodes in a bag node have to follow the same schema, so f is the same for all instances (subtrees), and the output space is of fixed dimension for all subtrees. We might use *max* or *mean* as an aggregation function g. Authors of the framework use one or more layers of a feedforward neural network as F.

4.3.3.3 Product model — $pm(f_1, \ldots, f_l, f_p)$

A product model consists of l submodels and one more final mapping f_p. Each of the submodels processes one child of a corresponding product data node. Each product data node can follow a different schema, so each f_i might differ (not like in the bag model). By applying submodels on the corresponding product nodes, we get l vectors with possibly different dimensions. The product model concatenates all vectors, and the result is transformed one or more times by f_p.

As all functions in the tree-structured model are at least piecewise differentiable with respect to all parameters, the backpropagation algorithm is adopted in the overall model optimization. We can find its adaptation in [68].

4.3.4 Modelling JSON documents using HMill framework

In the previous chapter, we described several approaches for *JSON* document modelling, here we move on with how the *HMill* framework deals with this problem. In [68], the authors presented experimental results for the *IoT device identification* use case, where the data was *JSON* documents.

Assuming that we have a set of *JSON* documents as a dataset $\mathcal{T}$ we require one condition to hold. All documents in our dataset have to follow the same structure (same schema), which means the following:

- Given a fixed position in the tree, we know which keys can be found in this particular position in all documents $d \in \mathcal{T}$.
- Values of the same key at the same position in all documents $d \in \mathcal{T}$ must follow the same structure.

- Arrays at the same position in all documents $d \in \mathcal{T}$ must be empty or contain the same structured objects.

Note that a document can follow the same schema even if it is missing some keys. The matching schema might seem too restrictive, but it is widespread that documents used for a single use case follow the same schema, especially machine-generated documents.

The *JSON* to *HMill* sample translation is defined on three basic abstract node types. Each leaf of the tree (*JSON* primitive data type) is transformed into an array data node $an(h(\dots))$, h denotes mapping to $\mathbb{R}^n$. The mapping h is identical for all leaves at the same position in the tree across the dataset, and it might be different for different positions. The product data node is used to model *JSON* object. Because the product node accepts an ordered set, the keys in the *JSON* object have to be ordered. Finally, *JSON* arrays are modelled as bag data nodes. The translation is demonstrated in the figure 4.4.

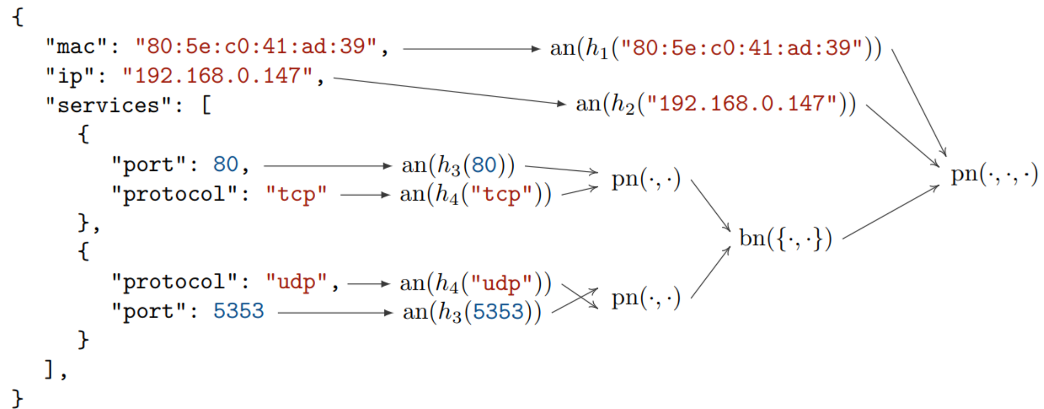


Figure 4.4: Translation of *JSON* document into *HMill* data nodes [68]

The authors of the framework stated that the model takes into account even the document's structure which exceeds the *flattening* approach. Compared to the rule-based approaches, the framework can learn more complex hypotheses because chosen aggregation can substitute various hand-defined rules. The *HMill* framework was used in the IoT device identification use case [68], which was a dataset of *JSON* documents modelled for multinomial classification. Reported accuracy is 96%. Authors experimented even in different setup for a general graph inference to demonstrate the generality of the *HMill* framework.

4.4 CAPEv2 classification

The thesis aims to create a classifier that classifies signatures based on behavioural features observed during the dynamic analysis.

4.4.1 Behavioural features

After a further look at the *behaviour* part of the original report, we decided to use *summary* and *enhanced* subtrees for our purpose. The reason is that other parts are pretty compre-

hensive, and we would not be able to train the model with the hardware resources we have. Those two are sufficient for a complete overview of malware behaviour as they include the essential features mentioned in chapter 2 (API calls, commands...). In *summary* part, we have unordered lists of these features, and, in *enhanced* part, there are sequences of actions with a timestamp.

If the model is still too complex to train under our conditions, we omit the *enhanced*, which is much sparser and longer than *summary*. We would lose the information about the order of events.

4.4.2 Signature classes

The signatures are usually assigned by the sandbox based on some atomic fact, such as API calls in the report. Let us call that *true cause* of signature, as it is considered ground truth, in this thesis. It can be one or more patterns seen in the behavioural report. The implementation of a signature is deterministically detecting its cause. If the cause is detected, the signature is added to the *JSON* report (see the whole process in 2.

We can consider the *copies self* signature example. If the same file as the analyzed one is among dropped files in the report, we will find *copies self* signature in the report.

An example of a signature entry in original *JSON* report could be seen in D. The most important aside from the *name* and *description* is the field *data* which sometimes contains the true cause.

4.4.2.1 Categorization

For modelling and model explaining, we categorize the signatures according to two factors. The first is the cause, which creates groups such as API calls, processes, dropped files, etc. The cause is determined according to the signature's implementation (in Python). In D we can see a simple example of implementation of *antidebugsetunhandledexceptionfilter* signature, which is just checking the presence of a specific API call.

The second categorization factor is the fact if the cause is directly presented among our features. For instance, if the signature implementation uses API calls, the API call list is directly in the log. If it uses the entropy of dropped files, it is not directly in the output.

The categorization may help us with the structure of the result, discussion, and further reasoning. We expect that models for different categories might have different accuracy.

Specific signatures for modelling will be chosen according to their frequency in the dataset. All details about the chosen signatures and even the mentioned groups will be described in the next part of the thesis.

4.4.3 Model

We want to train a binary classifier that predicts the presence of a particular signature according to behavioural features.

Hmill framework API is designed generally, so it is not accepting *JSON* documents directly. For this purpose *JsonGrinder* library might be used. It accepts an array of *JSON* documents

and produces *HMill* schema. The schema is then used to create *HMill* abstract data node tree and model tree. Example of a schema, and implied *HMill* model for our data is in the attachment (H).

Chapter 5

Model explaining

In the following chapter, we describe the model explainability definition and techniques used in practice. Finally, we give details on *HMill* explainer, which we use in our experiments. At the very end, we describe our use case of signature's binary classifier explanation.

In the case of complex models, we often want to explain their predictions to be sure about their reliability. The model explainability is crucial in critical systems, e.g., health, transport. If we want to use our models and want people to believe them, we should explain their predictions and demonstrate that they are not based on random correlations in the training data. For example, in [78], authors identified that their model classifies mainly according to the timestamp field in the original sample, which was different for malware and cleanware samples. That was an obvious mistake because this detail is not the difference between malware and cleanware but between analysis conditions.

In May 2018, General Data Protection Regulation (GDPR) became law. It has innovative clauses on automated decision-making and, to some extent, even the right of its explanation. All individuals might enforce to obtain "meaningful explanations of the logic involved" when automated decision making takes place [42]. That is a significant scientific challenge in the field where we face such a great boom regarding the statistical model's performance and a disproportionately weak understanding of its behaviour. As an example, we can see the neural net generalization, which is still very challenging for us [108].

Techniques of interpretation and explaining are growing in popularity as a tool for further statistical model analysis. It might lead us to better model understanding or shed some more light on the examined domain (extract new knowledge) [71].

5.1 Definition

Based on [71] we define two essential terms.

Definition 1 *An* iterpretation *is the mapping $\mathcal{Y} \to \mathcal{D}$, where $\mathcal{Y}$ is a* state space *(defined in 3, e.g., real-valued vectors or sequences, $\mathcal{D}$ denotes a domain which is human-readable and understandable (image, heatmap, sequence of words, etc.).*

Definition 2 *An* explanation *$e \in \mathcal{X}$ is a subset e of sample x that contributed for the predicted state $h(x) = \hat{y} \in \mathcal{Y}$, or contributed significantly more than other members of x.*

We often express the explanation as the original feature vector with a relevance score vector, e.g., real-valued vector of the same dimension as the input, where positive items indicate relevant features and zeros indicate irrelevant ones [71].

This definition of the *interpretation* is quite vague because human readability and understandability is not something to measure or observe precisely. On the other hand, the explanation is a little bit more specific. For example, we can see the task where we aim to select part of the original feature vector responsible for most probability accumulated in the output, e.g., softmax output. This output might be interpretable by human, or we have to find another mapping to an interpretable domain, e.g., $\mathbb{R}^n \to \mathcal{X}$. For instance, if we get an explanation of network classifying images, we might get a real-valued matrix. However, we translate it back to the image with highlighted pixels to be better understood. [63, 94, 62]. If our subject would be natural language processing, the explanation might be a highlighted text [9, 65].

A different perspective of understandability is looking at the available time that the user (human) is available or allowed to spend on the explanation understanding [42]. Then we can check the complexity or quality of the explanation by measuring the time to understand the explanation. However, we have to control the person's background knowledge performing the interpretation [42].

Authors of [71] address that the primary bias in *interpretability* is that the majority of the model evaluation metrics works with the model as with a black box. Sufficient information for the evaluation is just a set of predicted labels (or class probabilities) and ground-truth labels. Often, we do not observe model parameters, hyperparameters, model hypothesis space, etc.

Another problem is that some situations are not easily transferable to a numerical form, e.g., real-valued vectors. Examples we can find in ethics or legality. Interpretability of such concepts is ambiguous even in the world without machine learning.

5.2 Categorization

Based on [42], and [66], there are several aspects according to which we can categorize the model explaining.

If we know the model we are explaining and investigate its functioning, parameters, and other details during the learning process, we speak about the white box explaining — often called *transparency*. In this case, the goal of explanation might be to answer the question *How does the model work?*. On the other hand, if we examine the model's output without considering what model is used and how it works, we speak about black box explaining — often called *post-hoc* explanation. In this case, the question is *What else can the model tell us?*.

Based on the scope of interpretation, we may distinguish two categories. *Global interpretability* means that we can interpret all predictions. We know interpretation f: $\forall y \in \mathcal{Y}$, $\exists i \in \mathcal{D}\ f : y \mapsto i$. In other words, the interpretation is mapping (or relation), and if the model is globally interpretable, this mapping is *serial* (also called *left-total*). On the other hand, *local interpretability* means that we can interpret only some predictions, so the relation is not *serial*.

5.3 Explanability desiderata

In [66], authors present a comprehensive insight into interpretability research and interpretable model properties. We list some of them below.

5.3.1 Trust

Trust is the first term which is very complicated even due to its own interpretation. We might build trust in the model's performance, so the better the model is, the more trustworthy. However, the accuracy of the model is not the only metric we should consider. The situation is more complex in the sense that we need to examine the whole context. By context, we mean the accuracy on specific examples, e.g., unseen examples or examples where people can classify with high precision. We should also evaluate the accuracy repeatedly in time.

5.3.2 Causality X Correlation

Correlation of two random variables X, Y is defined in (5.1), where $cov(X, Y)$ denotes covariance and σ standard deviation. On the other hand, causality is defined as the relation between X (a cause), which contributes to the production of Y (an effect), X, Y might be events, processes, states, objects or generally random variables.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y} \tag{5.1}$$

Every statistician was instructed about situations where the data show us the results we want to see. That might be demonstrated in examples that should be so absurd that nobody can take it seriously. For example, the divorce rate in Maine correlates with the per capita consumption of margarine between 2000 and 2009 [1]. Those correlations are often called *spurious*. However, we have to remember that the underlying process generating the data is assumed to be random no matter how complex it can be. We must not forget that the researcher is making assumptions and choosing what data are modelled. The statistical model itself should not serve as an argument for the cause and effect relation between modelled variables. If we want to conclude causality, we should involve other experiments and research in the particular domain to uncover the generating process itself. On the other hand, the correlation is easily measurable, and a conclusion about it is based only on its calculation. More on this topic can be found in [55].

Even if we observe the correlation and know that the relationship is not random, it is difficult to conclude the cause and effect. One of the possible reasons might be a confounding variable. It is a variable influencing both features and states of our model. If such a variable exists, we might falsely conclude causality, although the confounder causes the correlation. More on this topic can be found in [96].

[1] http://www.tylervigen.com/spurious-correlations

5.3.3 Transferability

In a usual setup, we split our data randomly and create a training and testing set. Then we estimate the generalization error by observing the difference between training and testing error. However, regarding the model deployment, we should observe its behaviour in practice. It might face different situations or, even worse, its deployment might influence the domain itself. The difference between training and real data might be caused by the solid assumptions we make but cannot meet. This trend refers to the robust statistics field where we face the problem of assumption violation [28].

5.3.4 Informativness

This point is about the model's ability to extend human intuition and knowledge by pointing out the most important parts of comprehensive inputs. It can also provide a stronger overview of the space we are examining, e.g., provide some similarity measure on our examples, which might be essential for gaining labelled data using unsupervised or semisupervised learning.

5.3.5 Fair and ethical decision making

If we want algorithms making autonomous decisions under our control and being of our interest, we need to interpret its decisions. That is a very significant issue because we need to deal with the fact that artificial intelligence is much more capable of making fast and precise decisions than humans. It is not clear if we do not degrade artificial intelligence capabilities by trying to understand it. However, it is necessary because we have already used it against each other, e.g., [12]. By adopting GDPR, we face this challenge even for legal reasons. The field of ethics in AI [92] is an inexhaustible well of challenges beginning with autonomous driving and ending with absolute manipulation of a mass of people.

5.4 Interpretable model

As we mentioned, we can distinguish two basic types of explanation according to our goal — *transparency* and *post-hoc*.

5.4.1 Transparency

Authors in [66] refer to several attributes which can be treated during a particular model's *transparency* research — *simulatability, decomposability* and *algorithmic transparency*.

By *simulatability* is meant that the model prediction can be simulated by a human in a reasonable amount of time given the model parameters and input example. This capability is closely connected with the model complexity. That might seem like it is about the type of model such that we can say that, e.g., decision trees are better interpretable than neural nets. The truth is that simpler models like linear regression or decision trees tend to be better interpretable, but it is because they are usually involved in straightforward use cases. On the

other hand, simulatability is strictly determined by a limited amount of human cognition. That leads us to conclude that a very complex decision tree is not more transparent than a lightweight neural net.

Decomposability stands for the ability that each input, parameter and calculation admits an intuitive explanation. The input interpretability throws out the game majority of models where dimensionality reduction and other feature engineering techniques are involved. The interpretation of parameters and calculations might be a human-readable description of decision tree nodes, and the opposite might be a large number of weights and biases in a neural net.

The last notion of *algorithmic transparency* is about observing the learning algorithm and its mathematical background. The algorithm has to be fully explorable using mathematical tools. For example, in the linear model, the shape of the error surface can be understood, and we can prove that the training process will converge to a unique solution. On the other hand, heuristic algorithms used in deep models like *stochastic gradient descent* can not be fully observed, and we cannot be sure about its adaptation in a new use case.

Examples of a model with a significant level of transparency are linear/logistic regression, decision trees, KNN, Rule-based learners, and generative additive models. More on this topic can be found in [10].

5.4.2 Post-hoc

Post-hoc interpretability has a different goal than the previous approach. It can extract more information from the model and help us gain new overall knowledge or understand what is in the input that causes such a prediction. This technique can be used to interpret opaque (not transparent) models without examining their complex logic.

There are several techniques for post-hoc interpretation. Their list follows.

- *Text explanation* — We still assume that we cannot assign textual interpretation by hand because that might mean that our model is transparent. However, we might automate the inference of a textual form. We can train another model which maps the prediction of the original model to its textual explanation. An example of such an approach is in [58], where authors trained a reinforcement model to perform a particular task and a second model to explain its decisions.
- *Visualization* — Images and other visual outputs are considered very straightforward for human understanding. We often involve dimensionality reduction and other techniques to display the situation in two or three dimensions, such as [77]. An example of a visualization of neural net explanation using heatmap can be found in [106].
- *Local explanations* — This approach explains specific parts of the training set, e.g., specific samples. It might find the parts of a sample that contribute to the prediction the most. An example might be the saliency map used for neural nets [94]. It is important to emphasize that the explanation has to be treated in a specific context, i.e., the saliency map might change drastically even if the example was changed only slightly.

- *Explanations by example* — Usually, if the teacher explains the theory with a running example, especially in mathematics, there is a much greater chance that many students would understand. The principle is the same in the model explanation by example. The model can provide such an explanation along with predictions, e.g., prediction and a set of examples which are similar [18]. We might involve a clustering algorithm.
- *Explanations by simplification* — This approach aims at model simplification while maintaining its performance.
- *Feature relevance explanation* — We aim at scoring the input variables, which are later compared based on their scores, and we can conclude which variables are the most important for a particular prediction.

This kind of explaining is used with SVM, where we can see model simplification or local explanations. Other significant models are neural nets and their variations, where we see feature relevance and visualization techniques. Examples can be found in [10].

Another example of a general model explaining is in [5], where authors focus on self-explaining models. Other sources might be [97, 86, 71].

5.5 Explaining HMill models

Our model can model tree-structured data — *JSON* documents, as stated in the previous chapter. This section describes an explanation method for *HMill* models proposed and demonstrated in [78]. Introduced explainer attempts to explain structured *HMill* model. It uses a post-hoc approach with *feature relevance explanation*. So far, it is the only known approach of *HMill* model explanation.

The goal of *HMill* explainer is to find a minimal subset of the input sample (*JSON* subtree), which is classified to the same class as the original sample. We can identify what parts of the original *JSON* are the most relevant. It might also improve our understanding of the domain-specific knowledge, as the authors state.

In the following text, we assume an *HMill* binary classification model being a black box decomposed only to two function h and f. The first function is $h(d) = v$ where d is *JSON* document or its subtree and the output $v \in \mathbb{R}^m$ represents the embedded sample (all abstract model nodes are evaluated but the root). The embedded sample is then classified by $f(h(d)) = c$, where f denotes final abstract model node evaluation function and c confidence that d belongs to the positive class.

5.5.1 Explainer steps

The basic idea builds on top of the subtree selection problem solution. Our problem is a specific case of subtree selection. It can be formulated such that for a given tree T, an expensive evaluation function $r(t) = q$ ($q \in \mathbb{R}$) and a threshold $\tau \in \mathbb{R}$, we want to find subtree t with minimal number of nodes such that $r(t) \geq \tau$.

In the case of *HMill* explainer, the input tree is a *JSON* document, and the evaluation function is f, which outputs the confidence that the document belongs to a positive class. The

authors introduced several subset selection methods, which is a less general case of subtree problem. It is not possible to adopt subset selection methods in subtree problems directly. Firstly, we have to decide if we need to maintain the tree structure of the result or not. The authors mention two solutions to deal with this problem: first, ignoring the structure and the second, exploiting it.

5.5.2 Subset selection

Greedy addition starts with an empty subset and in each step adds a new item until the threshold is achieved. Each new item is chosen such that the gain in the evaluation function is maximal across all elements which are not in the subset.

Heuristic addition sorts the elements in the set by heuristic ranking (see below) and adds elements to the subset starting from the largest ranking until the threshold is achieved.

Random removal might follow any of the previous methods. It starts with the initial subset, which has already achieved the threshold. It permutes all items in the subset and removes items from the beginning until the evaluation drops below the threshold. If it drops, the lastly removed item is added back, and the algorithm continues with a new permutation.

5.5.3 Minimal subtree adaptation

Flat search performs subset selection on isolated nodes of the tree. The root is added to the explanation by default. After subset selection is made, all nodes which are not reachable from the root are removed from the explanation. That can be done because they do not impact classification based on the definition of the *HMill* model tree structure.

Level-by-level search performs subset selection on each level of the tree. It takes into account only nodes whose parents are in the explanation so far.

5.5.4 Subtree ranking

5.5.4.1 Model gradient ranking

This approach is based on the absolute value of gradients for the parts of the input. That is adopted even in the case of a saliency map of image processing neural net. The crucial idea is to compute the gradient vector ((5.2)) of the model with respect to the embedding of a particular subtree c of a particular sample d. Alternatively, we want to examine how much a slight change in the embedding of a specific subtree influences the model's prediction. Computation of the gradient ranking of a particular subtree is the absolute value of the sum of items in the resulting gradient vector. Note that $h(d)$ is originally a function of $h(c)$, which is not obvious from (5.2). However, if it was not, the derivative would be zero.

$$\frac{\partial f(h(d))}{\partial h(c)} \in \mathbb{R}^m \tag{5.2}$$

5.5.4.2 GNN explainer mask ranking

This method is originally designed to explain *GNN* models [104]. The main idea is to use this explainer for the graph created by *JSON* subtrees and edges between them. Explainer uses a mask $m = \mathbb{R}^{|\mathcal{E}|}$, where $|\mathcal{E}|$ denotes all edges between the subtrees and their predecessors in a *JSON* document and $m_i \in [0, 1]$. This mask represents how much information is passed along each edge during the update step of GNNs. Suppose the explainer is asked to explain a classification decision on a particular subtree. In that case, the mask is optimized using stochastic gradient descent to maximize the probability of correct classification on that subtree. After the optimization, the explainer suggests k edges with the largest values as an explanation. The value of k should be properly tuned because the explanation does not have to be classified to the same class as the original sample if k is fixed in advance. The values of the mask might be used as ranking in *HMill* explainer.

5.5.4.3 Banzhaf values

Game theoretical approaches in feature extraction which uses *Shapley values* and *Banhaf values* can be found in [2]. Both come from the cooperative game theory. They are related to the metric of how much a certain player contributes to various coalitions on average. In feature extraction, the Banzhaf value might describe interdependency among the extracted features and their relevancy to the target class. On the other hand, *shapley value* might be used to show the contribution of a particular feature in improving the classification accuracy when all possible coalitions of features are considered. In the case of *HMill* explainer, the authors used Bazhaf values.

A sampling algorithm is used to approximate Banzhaf values [11]. There are two values stored for each subtree in the *JSON* document – the classifier's average confidence in coalitions which includes the subtree and the average confidence of coalitions that do not. Coalitions are generated randomly. After running more iterations, the approximation of Banzhaf values for each subtree is the difference between the two values stored in it. Banzhaf values approximation is used as the subtree ranking in *HMill* explainer. If we do not fix the seed, the explanation is stochastic and might be unstable regarding the number of iterations and the output.

5.5.5 Results

Authors in [78] present qualitative and quantitative analysis for different *HMill* explainer setups. The best result regarding the computational time and size of explanation was reported for the Banzhaf values-based heuristic addition approach combined with Level-by-level search. We will use this setup.

5.6 Other methods for structured data

We mentioned three alternative techniques for *JSON* modelling in the previous chapters — rules, flattening, and GNNs. Flattening is a technique of feature engineering more than

modelling. It might be challenging to explain its predictions, as a general explanation algorithm stops at the flattened features and cannot explain the flattening functions. That refers to the weak decomposability of such methods mentioned earlier.

Rules are usually transparent, so their interpretability is straightforward. Finally, examples of graph neural net explaining can be found in [104, 47]

5.7 CAPEv2 explaining

In the chapter 3, we introduced two factors according to which we categorize signatures.

We expect that explanation of the model, the cause of which is in the report, should be a set containing this cause. As an example, we expect that if the original signature's cause is a specific API call, it should be presented in the explanation of the binary classifier for this signature. As authors of *HMill* explainer mentioned, we hope that the explanation contains even something new. In other words, we expect that we could observe explanations that contain entries that are not the original cause. However, they might reasonably substitute it — they are connected to the same effects. It is also possible to identify new signatures because all samples are malicious and the model might generalize based on different similarities in the training set.

We want to explain at least one hundred samples for each classifier to observe repeatedly seen parts. We choose positive samples because we want to explain the positive class predictions.

5.7.1 Additions

Since *HMill* explainer usually process sample by sample, one hundred explanations mean one hundred *JSON* files. This quantity is still hard to interpret, and that is why we involved three additional ideas.

We merged all explanations for one signature into a one *JSON* file and for each entry computed a number of occurrences across explanations. We assume that the most general formulation of an explanation should be seen repeatedly.

We also counted the frequency of each particular key (name of the field in JSON file, e.g., *read files, resolved apis*) in explanations such that for each signature, we see how often the explainer detects a particular key. We can compare the original signature's cause with the most seen key.

The last idea is that we compute the frequencies of entries seen across different signatures. We assume that in such a way, we could partially identify the bias that is caused by the entries that we see in multiple explanations across signatures. In such a case, it should be considered too general.

Part II

Experiments

Chapter 6

Infrastucture and data collection

This chapter describes the realization of the data collection process using *CAPEv2* sandbox. The sandbox is described in the chapter 2, here we focus on the specific setup, problems we experienced and their solution. We have a data source of malware samples MalwareBazaar mentioned in the thesis introduction. The output of this task is a dataset of dynamic malware analyses. It includes behavioural features and signatures, both input for our *HMill* model.

Although this chapter is shorter than the previous ones, we spent the most significant time on this task. All scripts and other outcomes are part of the attachment (H), and the most important tools are listed in G.

6.1 Host machine

The host machine is where the sandbox environment and virtualization software is running. We know that an analysis of one malware sample takes up to five minutes, and we want to have as many samples as possible. That is why we want to run several distributed host machines.

The whole process of a host machine initialization is automated to be able to set it up multiple times. The initialization consists of several steps:

- Install host operating system — recommended Ubuntu
- Enable SSH to be able to access it remotely
- Enable basic security — firewall and supporting tools
- Install virtualization software — recommended KVM QEMU
- Copy virtual machine images to the host
- Sandbox initialization and configuration
- Data collection script initialization

To run all steps on multiple machines at once, we used Ansible, a network orchestration tool. We also added some usual management functions, e.g., to copy new configurations from the server or clean up the sandbox data.

Hardware resources on host machines are 256GB SSD and 16GB RAM, which means that a sufficient number of virtual machines per one host is four. We experimented with more, but there was an overload which might lead to biased analysis results.

Various issues accompanied the automation of the whole process. Such a process often needs manual steps, and their automation is very challenging. Issues were caused mainly by our low experience, and sometimes poor documentation was involved. Especially the process of virtual machine images creation and copying was connected with issues. Overall, we have configured seven host machines.

6.2 Guest machine

Guest denotes the virtual machine where the malware sample runs and where the *CAPEv2* monitor operates. We used Windows 7 as an operating system. The crucial goal of the guest machine is to look like an ordinary computer that is in regular use. Due to the virtualization usage, we had to care about the sandbox evasion techniques mentioned in 2.

There are two options for anti-evasion setup in *CAPEv2* sandbox. Firstly, we experimented with vmcloak[1]. We were able to run and use this tool. However, it supports only VirtualBox, which is not recommended by *CAPEv2* because of its performance. The project is also no longer maintained, and some functions did not work, e.g., taking snapshots. The second option is to use a script recommended by sandbox contributors[2] and perform manual steps in the virtual machine configuration[3]. After several unsuccessful attempts with low-level virtual machine misconfigurations, we were able to create four working images.

The sandbox requires disabling the firewall and running Python on the guest machine. We added the most popular applications like Google Chrome, Firefox, Adobe reader, Spotify. We added one private key to C://Users/Administrator/.ssh and one password to the Google Chrome password database. We downloaded random images and documents from the internet.

6.3 Network setup

The network setup is a crucial point in the dynamic analysis. The guest machine has to reach the host machine to stay in touch with the result server. Secondly, there is an internet connection for the guest, which might be necessary for some malware types. As an example, we can see *dropper*, which is responsible for downloading a payload.

We decided to collect the data under two different conditions — with an internet connection (denoted by *internet*) and without (denoted by *none*). We engaged both because the *internet* is much more difficult to set up and secure, and we wanted to start data collection as soon as possible. Both architectures are figured in the appendix C.

[1]https://github.com/hatching/vmcloak

[2]https://github.com/doomedraven/Tools/blob/master/Virtualization/kvm-qemu.sh

[3]https://www.doomedraven.com/2016/05/kvm.html#modifying-kvm-qemu-kvm-settings-for-malware-analysis

6.3.1 None

None setup is a straightforward option for an isolated network between host and guest. This approach requires a host-only interface created in *KVM* virtualization tool. Host and guest are assigned with IP addresses from the same range. From a security point of view, we have to set up a firewall on the host machine. It should accept connections from the isolated network only on the result server's port.

6.3.2 Internet

We want to provide internet access to the guest machine during the analysis run. For this purpose, we prepared a VPN connection to the secured network through which the communication should be forwarded. That is considered a good practice to observe what the samples are doing and be able to stop it fastly. We call this network *dirty lab*.

CAPEv2 supports VPN connection setup for each guest machine separately. We knew that it would be better to have a central gateway for all local traffic than connecting each particular guest machines to the *dirty lab*. However, we decided to use native sandbox functions at first. After experiencing some issues with *CAPEv2* VPN configuration and an unsuccessful issue reporting, we decided to find another custom way.

The main requirement is to centralize the traffic from the local network (malicious) to *dirty lab*. The exit point we call *router*. The surrounding university network has to be secured and isolated from malicious traffic. We also need to monitor host machines because of a potential intrusion and centralize logs from host machines.

In the following text we use *l2* and *l3* as a designation referring to ISO/OSI model of network communication [109]. By *l2* we mean communication on the data link layer. Specifically, we mean ethernet/802.11 local networks where MAC (Media access address) is used for device identification. By *l3* we mean IP communication on the network layer. IP addresses are used for device identification on *l3*.

An expert recommended the designed network architecture after a consultation. Its basic idea is to avoid *l3* communication on the local network such that IP address from one range is assigned only to the guest machine and then to the router device. This idea was supported mainly by the *l2* VPN on the local network.

The role of the router is to receive the traffic going out of the local network and forward it to the *dirty lab*. It can also monitor and capture the traffic. In *dirty lab*, we were provided with the ipv4 interface only, so the *router* performs network address translation (NAT). *Router* machine is running Ubuntu operating system. It is configured to connect to *dirty lab* using *l3* VPN and to the guest machines using *l2* VPN. In *l2* VPN, *router* is a server and in *l3* VPN, it is a client (server is running in the *dirty lab*). *Router* is realized as a virtual machine with fast recovery capability. All logs from the server are sent to the central machine.

The host machine has to be configured as a client in communication with *router* using *l2* VPN to forward the guest's traffic through it. The idea of this setup is that all the traffic leaving this device is encapsulated in packets with university network IP addresses. This network is unknown to the guest device where malware is running. From the guest's point of view, the connection between the *router* and a *guest* is on *l2*.

On the host, the interface for *l2* VPN communication (TAP) is bridged with the original interface, which allows communication with the guest virtual machine (originally host-only).

A connection to the internet from the guest machine goes through the host machine, the router, and the dirty lab to the internet.

There has to be an additional setup on the host machine besides that listed in the case of *none*. Each host machine in the distributed cluster has to send all sandbox logs to the central Syslog server. All machines have to be set up with a monitoring tool to detect intrusion[4].

6.4 Data collection pipeline

During the distributed data collection, we used the following terminology. *Master* is a machine responsible for sample distribution, and it has access to a NAS, where it stores analysis results. *Worker* is another name for *host* machine in the context of distribution.

This section describes the whole process of data collection, beginning with a malware sample going over its dynamic analysis and ending with the behavioural features and signatures extracted from the *JSON* report. All programs implemented to solve the mentioned problems are part of the attachment (H). Particular steps are automated, and their description follows.

1. Download samples from the *malwareBazaar*
2. Filter PE files only
3. Retrieve additional metadata
4. Add hashes to database
5. Distribute samples to workers
6. Analyze samples and send results back
7. Store results
8. Extract JSON reports
9. Prune unnecessary parts

6.4.1 Abuse.ch MalwareBazaar

The place where we downloaded malware samples was abuse.ch[5] specifically part called *Malwarebazaar*. The reason for its usage is free access without any claims. *MalwareBazaar* is a database of malicious (no benign files or adware) samples that anybody can share and download. In May of 2021, it contains over 325 000 samples. Malware samples are downloaded in the compressed form — one archive for every day since the start of the *MalwareBazaar* project.

[4] example https://aide.github.io/

[5] https://abuse.ch/

6.4.2 File filtering

Our project aims only at PE files described in 2. After decompression of the original archives, we filter the files based on the file extension and file headers. We also filter the compressed files, decompress them, and again filter PE files only.

6.4.3 Metadata

The secondary intention was to obtain some basic metadata about each file to have basic information. We were able to gain academic access to the VirusTotal API[6]. We downloaded a metadata report for each of our samples. The report contains basic static information like hashes and fuzzy hashes, extracted strings, detection of various antiviruses, and even a summary of reports of used sandboxes.

6.4.4 Distributed sandbox

After dealing with issues in the host initialization part and even in the VPN setup part, we also encountered issues while setting up the distributed sandbox. *CAPEv2* can orchestrate multiple host machines. It uses distributed mongo database[7] combined with a script that is run on the master machine to check the connection to the registered worker devices. We spent with the configuration of distributed *CAPEv2* large amount of time trying multiple different ways and following several pieces of advice but unsuccessfully. We decided to implement our lightweight solution for time reasons.

Hashes of our files are stored in a database (in our case *JSON* file) with additional attributes. A script runs on the master machine that distributes samples among worker machines using their REST API. After the analysis is done, another script on the worker machine compresses the result and sends it back to the master. The last script manages the coming results and saves them to the NAS. Everything is recorded in the database.

6.4.5 Result postprocessing

We need only part of the *JSON* report, specifically behavioural features and signatures, for further modelling. Its extraction was done in two steps. Firstly, we decompressed the analysis result and extracted the *JSON* report only. Secondly, we extracted the mentioned parts and saved the shrank output. The whole report might have even gigabytes. However, the shrank variant has usually tens of megabytes. We could transfer the output to the metacentre where the model computation and explanation took place.

6.5 Collected dataset

When we start modelling experiments, we have a dataset consisting of 80,000 different samples in *none* network setup. The *internet* configuration took us several weeks to deal with,

[6]https://developers.virustotal.com/v3.0/reference
[7]https://www.mongodb.com/

and the data collection started later and was slower. The *internet* data will be investigated in future work as its collection continues.

The complete dataset has approximately 2,5 TB in compressed form. Extracted features and signatures are much smaller (tens of gigabytes).

Not all outputs of the sandbox described in 2 are presented in each analysis result because the configuration and other conditions influence it.

After examining the histogram of seen signatures, we chose a subset based on their frequency in the training set. We prefered signatures that are implemented in Python for convenient investigation of the original cause. In D, our candidates can be seen — their frequencies in the dataset, and additional information, including even the groups described in the chapter 3.

Chapter 7

Results

In this chapter, we cover our experiments in *HMill* modelling and explaining, their setup, results and discussion. We have a dataset of *JSON* reports containing *behaviour* part and *signatures*.

7.1 Model

We described our motivation in the first part of the thesis. Details about modelling are mainly at the end of chapter 4. Here we summarize only experiments.

7.1.1 Details

7.1.1.1 Hyperparameters

We build on previous experiments in [68] where the authors published even used hyperparameters in the *device identification* example, which we used as our initial setup.

In the table 7.1, we can see the model's hyperparameters and other training-independent facts that we used.

Table 7.1: Hyperparameters of *HMill* model

Parameter	**Value**
samples	80000 (1:1 testing and training)
minibatch size	1000
hidden units (neurons)	20
iterations	120
optimizer	*ADAM* [57]
loss function	*logit cross entropy* (3)

The difference of our parameters from [68] is *minibatch size* and *number of iterations*. Data used in their case were of a much smaller scale which could be the reason for the larger *minibatch size*. The smaller *number of iterations* is caused by resource and time limitations. We checked overfitting by monitoring the difference between accuracy on training and testing data. They did not differ significantly.

7.1.1.2 Experiments

We performed experiments with different feature sets. The first experiment used both *enhanced* and *summary* parts as an input vector for the model. Due to the input size, the model was too large to converge to some significant accuracy. We were not able to train it on hardware and with thread limitations in a feasible time. This reality led us to skip the larger part of the feature vector — *enhanced* part. This part contains a series of events with many redundancies and additional data, e.g., function parameters. Each event has its own object, so the information is much sparser than in the *summary* part.

The resulting model is working with *summary* part only. We expected that could happen as we described in 4. The tighter feature vector was an advantage for the training time.

For evaluation, we chose the metrics following the chapter 3 where we mentioned classifiers in the cybersecurity field and their pitfalls.

7.1.1.3 Technicalities

In the G, we describe the technical background for *HMill* model training and evaluation, such as libraries and programming languages. We unsuccessfully experimented with multi-threaded gradient computation. The resulting model was trained on one CPU, which shrank our possibilities a little. However, basic linear algebra subprograms (BLAS) were involved in the matrix multiplication and were multi-threaded.

The code of the model is in the thesis attachment H.

7.1.2 Results

Results of the experiment could be seen in 7.2. The table is divided into two groups based on the categories presented in 4. In the table, we also include a percentage of positive examples for the reader to see the balance of the dataset and assess the *FNR* and *FPR*.

There are also some other metrics and visualizations in E.

7.1.3 Discussion

Signature with the cause in report

In the first category of signatures, we observe quite consistent balanced accuracy above 95 %. Signature *copiesself* has 92 %, which is still sufficient for explanation. This deviation could be caused by the fact that the original signature is examining dropped files and checking if the analyzed file is among them. Nevertheless, the filename varies, so the entropy

Table 7.2: Model performance (values are rounded off to three decimal digits, P denotes positive examples ratio in our dataset)

Signature	**Bal. acc.**	**FNR**	**FPR**	**P [%]**
antidebug setunhandledexceptionfilter	0.9801	0.0289	0.0109	45
copiesself	0.924	0.125	0.0279	18
deletesself	0.997	0.005	0.002	27
enumeratesrunningprocesses	0.972	0.050	0.007	16
stealthtimeout	0.701	0.064	0.331	21
useswindowsutilities	0.958	0.078	0.006	18
removeszoneidads	0.999	0.000	0.000	28
antisandboxsleep	0.969	0.037	0.026	39
dropper	0.911	0.147	0.032	15
invalidauthenticodesignature	0.607	0.668	0.113	36
packerentropy	0.605	0.748	0.043	22
stealthnetwork	0.942	0.008	0.109	66

might be very high and might cause big *FNR*. Signature *deletesself* might have similar issues, but it is determined according to API calls and not dropped files which could cause that its accuracy is better than the previous. The only significant outlier in this group is *stealthtimeout* signature which examines a sequence of API calls that could be quite complicated. After going through some files where this signature was presented, we could not identify the particular API calls in the original log even by hand. Its prediction might be more tricky. The main cause of low balanced accuracy is, in this case, *FPR*.

Despite a single anomaly, this signature group shows reasonable accuracy, as we expected.

Signature without the cause in report

The second group behaves more unexpectedly because the accuracy in three of five cases is above 90 %. Signature *antisandboxsleep* uses API calls in a more complex way, so the original classifier may involve these. *Dropper* has significantly larger *FNR*. However, its overall accuracy is still high. *Stealthnetwork* should look at the network activity, which is not among the features. The excellent accuracy of these models arouses our interest in the explaining part. In cases of *invalidauthenticodesignature* and *packerentropy*, the accuracy is significantly lower than in other cases, as we expected. Overall, the first group has statistically better accuracy than the second one, as we expected.

We also performed several experiments with a more general multilabel classifier, but we did not observe convergence with our computational resources.

This part concludes that we can train *HMill* classifier to classify the presence of a particular signature based on the summary part of the behavioural report from *CAPEv2* sandbox.

This classifier has a significant accuracy of more than 90% as balanced accuracy for most selected signatures, which is sufficient for further explaining experiments because such models should have strong confidence.

7.2 Explainer

7.2.1 Details

The motivation and expectations regarding the model explaining are described in 5.

We performed two explaining experiments using *ExplainMill.jl* (described in G). We explained all models with a balanced accuracy above 70 %. The rest is not relevant due to its low overall confidence.

We used *Banzhaf* values as a subtree ranking method followed by the *Heuristic addition* subset selection. *Level-by-level* search was used as an adaptation for the minimal subtree problem. *Random removal* was also involved.

Explainer code is in attachments (H). We used a similar setup as authors of the tool [78]. We extracted several examples from the testing set in each run. We attempted to explain only positive examples, which were truly classified into the positive class with confidence above the specified threshold. The confidence threshold we used is 0.99 for the first run and 0.9 for the second run. We decreased it by 0.1 if no results were found in the data subset. We run the explainer on each of the chosen examples separately. We also used our additions described in chapter 5.

The number of explanations may vary because of the difference between the confidence levels of models. In the second run, we attempted to normalize the number of explanations to be 100 per signature, but we still were not successful in some cases. In F, we can see the number of explanations for both runs and other details.

7.2.2 Results

All original outputs and additional aggregations are in attachments (H) — merged explanations are in *merged* directory, frequencies of keys are in *freq.json* and merged keys across the signatures are in *overall.json*. Some statistics about the explanations can be found in F.

7.2.3 Discussion

The size of the original JSON file with only the behavioural part can be hundreds but even thousands of entries (average is around 3000 but included even the signature part). The average size of the explanation is 3–5 entries.

We formulated the following discussion of the results after presenting them to an expert. We formulate assumptions or hypotheses because we have to anticipate the risks mentioned in the chapter 5, especially the *causality X correlation* problem and the *confounding variable existence*. We are aiming at the observation description more than concluding.

antidebug setunhandledexceptionfilter

The most seen keys are *read keys, resolved APIs, executed commands*. It includes even API calls, which are the signature cause. Among entries, the most seen are *kernel32.dll.IsProcessor FeaturePresent* (153/377) API and *DisableUserModeCallbackFilter* (34/377) registry key. Those are presented in other explanations once and twice, so it does not look like something too general but also not unique. The registry key is related to exceptions, and the original API call is also related to them.

copies self

The most seen keys in explanations are *write files, executed commands, delete files* and the first is seen in all explanations, and it also coincides with the original cause, which might be a clue that the model uses what is expected, and its generalization goes the right way. Among entries the most seen are *ikkzowxr.exe* (13/100) file, *WerFault.exe* (13/100) file and *StikyNot yakuza* mutex. The first file is prevalent across different signatures. The mutex is also seen more than one time in explanations.

deletes self

The most seen keys in explanations are *deleted files, write files, executed commands*. The first is seen in all explanations. Here we can see some generalization because the original signature does not check the deleted files directly. It uses the API call to detect the file removal. However, the model uses deleted files with high accuracy. We also checked if this trend is not seen in more cases, but this is unique that all explanations include deleted files. This example is unique because the causal relation is straightforward, i.e., the API call causes that the file is deleted, and it appears among the deleted files.

enumerates running processes

The most seen keys in explanations are *executed commands, mutexes, read keys*. These do not include the original cause, which was the API call. Among entries, the most seen is "*IESQMMUTEX0208*" (17/84) mutex, but this mutex is quite common. The accuracy of this classifier is significant, but we cannot generalize to a more specific subset using our explanations.

stealth timeout

The most seen keys in explanations are *executed commands, files* which does not include the original cause — API call. The most seen entry is *DisableUserModeCallbackFilter* (11/78) registry. Nevertheless, the situation is the same as in the previous case. We are not able to generalize more.

uses windows utilities

The most seen key in explanations is *executed commands* which is included in each explanation, and it coincides with the original cause. The most frequent commands are *netsh, schtasks.exe*. This case is another clue that the model uses what is expected, and its generalization might go the right way.

removes zoneid ads

The most seen keys are *delete files, keys*. The first is seen in each explanation, but the original signature is using API calls. We are not able to identify specific redundant entries, but we identified one great conformity. The original signature implementation includes following *.endswith(":Zone.Identifier")* so it is detecting end of API call argument and even *.startswith("DeleteFile")* is detecting the name of API starting with a specific string. These two facts perfectly correlate with our explanations, where the majority of deleted files includes *:Zone.Identifier* suffix.

antisandbox sleep

The most seen keys in explanations are *write keys, keys, read keys* which does not correspond to the original cause. The most seen entry is *HKEY CURRENT USER/...* (63/100) registry key. We see this registry key in the case of two signatures. We do not see a direct relation between this key and the original cause.

dropper

The most seen keys in explanations are *write files, executed commands, mutexes*. The first is presented in all explanations. The second is not only in negligible fraction. The original cause is not trivial but dropped files are there, which corresponds to the first key. Among entries, the most seen is *IESQMMUTEX0208* (28/71) mutex, but this mutex was mentioned earlier as too general.

stealth network

The most seen keys in explanations are *keys, files*. The original cause (network) is not presented in the input at all. It looks like registry keys play a significant role. However, neither in the case of registry keys we can not find specific redundant entries.

In particular cases, we can see several situations. Sometimes the model explanations correspond to the original cause. That is a clue that the model uses what we expected, and its generalization might go the right way (e.g., *copies self*). There is even a particular case where the original cause does not fit, but the explanations logically correlate with it. In the case of *deletes self*, we see key *deleted files* in all cases. However, the original signature is detecting the same thing, but according to *api calls*. This example is unique because the causality is straightforward. The API call causes that the file is deleted, and it appears among

deleted files. However, the model generalized to that which should not be overlooked. A different case is *stealth network* where we do not see a direct cause of the fact that registry keys are often used in the explanation, even though they are not the original cause.

Choosing the most used key is one way, but the second is investigating particular entries (specific calls, files, mutexes...). It is challenging to interpret them and connect them to specific causes because their variance is enormous, as we expected. The most significant observation is in *removes zoneid ads*. We can see that the model mainly uses the same entries as the original signature (with the same suffix). That should also be considered as a clue that the model generalizes the right way.

Using our method, we were able to identify too general parts of explanations. We can see mutexes that are presented very often across different model's explanations. They might be considered confounding variables, as well as some files that are repeatedly seen in reports. Both play a significant role in detecting a particular family or classifying malware/cleanware, but they should not be used to identify particular behaviour.

In several cases, we cannot identify any direct cause of the model's high accuracy (e.g., *antisandbox sleep*) because the explanation is ambiguous. The reason might be a spurious correlation mentioned in the chapter 5. Without a more extensive dataset or some methods for causality detection, this might be impossible to discover.

After organizing theory in 5, we are cautious. Explaining is a complicated field with many challenges. We can not be sure about the output, especially using *post hoc* explanation per sample. The computation of *Banzhaf values* and randomness of the input causes the explanation to be a random variable as well.

Nevertheless, our observations indicate that some models strongly involve original causes in their predictions. That leads us to future work where the main interest should be improving the aggregation of particular explanations, detecting too general concepts (across classes), and confounding variable detection. It is noteworthy that our *post hoc* explanation should perform better with more extensive datasets. However, also the *transparency* approaches should be taken into account.

Suppose we can make the explanation more accessible to the client, e.g., a security engineer. In that case, there is a significant chance for *HMill* models to be used during malware analysis in real-world applications. The main reason is their high accuracy and their ability to process standard data formats like *JSON* document and provide an explanation in the same form. Of course, more complex examples, broader datasets, and further testing have to be involved.

Chapter 8

Conclusions

The main objective of this thesis was to design a pipeline that has a malware dataset as the input and a machine learning model and its explanation as the output. The whole process was motivated by high accuracy model interpretability to achieve greater compliance of machine learning and cybersecurity. Theory background and methods are summarized in the first part of the thesis. The setup, experienced problems, results, and their discussion are in the second part.

We set up eight physical machines with the *CAPEv2* sandbox in two different setups — with internet and without internet connection. Using the open source sandbox and our programs, we collected dynamic malware analyses for 80,000 malware samples retrieved from MalwareBazaar[1]. We reported the problems experienced during the data collection process and the description of the whole setup, including our code.

We used *JSON* reports of the sandbox as an input for *Hierarchical multiple instance learning* framework [68], the choice of this technology was justified by its ability to model *JSON* documents and better scalability in comparison with other methods. The classification model features are behavioural parts, and predicted classes are malware signatures, both included in the original *JSON* report. To evaluate our models better, we investigated the original signature's implementation and found out their true cause. We created a binary classifier for each of the chosen signatures (overall 12). We observed how each model performs in the context of the true signature's cause. Nine classifiers had a balanced accuracy of more than 90%. We reported and discussed individual results.

Finally, we experimented with the model explaining. Even though there might be hundreds of entries from the original behavioural report used as a feature set, the explainer only provides 3–5 entries as an explanation for each of the nine explained models. It is evident from our observations that some models were intensely associated with the original signature's cause. It is worth noting that there were cases where the model used different behavioural features with high accuracy. We reported and discussed all results.

Despite the significant amount of work we faced during the sandboxing, we managed to meet the goals of this thesis. We wanted our experiments to be repeatable, and therefore the source code and other technicalities are in the attachment of the thesis. In addition, we mentioned specific issues faced during the work, along with ideas for future work.

[1]https://bazaar.abuse.ch/

Future work

The sandbox *CAPEv2* does not include native support for extensive data collection since it is designed as a tool for malware analysis more than for machine learning experiments. Because our solution consists of many manual steps, it is worth adding some out-of-box solution for the clustered sandbox run to collect larger datasets. This solution might be based on the existing parts of the sandbox, which did not work for us. It can also be built on top of our lightweight tools. The main objective is to make it more user friendly for everyday use in machine learning.

HMill models showed good accuracy, and the framework should be part of other experiments with complex data like ours. There should also be larger datasets of dynamic analysis reports with different signatures and malware samples, such as the one with the *internet* data which we did not use eventually. Multilabel classification might be involved in the signature prediction as well.

The data quality in the cybersecurity domain should not be overlooked, e.g., an additional effort in redundancy/noise reduction in reports. That is also related to precisely controlled conditions during the malware analysis.

The framework creates the model directly from the *JSON* data which is very convenient for the model explanation. We can retrieve the explanation directly as a human-readable *JSON* document. This capability should be examined in different situations, emphasizing practical applications, e.g., new signatures extraction, zero-days, and interpretability for cybersecurity professionals. Addressed challenges like causality detection or spurious correlation should also be taken into account.

Appendix A

Classifier evaluation metrics

Table A.1: Classifier evaluation metrics

Metric	**Formula**	**Description**
accuracy	$\frac{TP+TN}{TP+TN+FN+FP}$	the ratio of correctly classified examples to all examples (imbalanced dataset may bias its interpretation)
false positive rate (FPR)	$\frac{FP}{FP+TN}$	the ratio of misclassified positive examples to all examples classified positive
false negative rate (FNR)	$\frac{FN}{FN+TP}$	the ratio of misclassified negative examples to all examples classified negative
true positive rate or recall (TPR)	$\frac{TP}{TP+FN}$	the ratio of truly classified positive examples to all positive examples
true negative rate (TNR)	$\frac{TN}{TN+FP}$	the ratio of truly classified negative examples to all negative examples
precision	$\frac{TP}{TP+FP}$	the ratio of truly classified positive examples to all examples classified as positive
balanced accuracy	$\frac{TNR+TPR}{2}$	shows average accuracy balanced for both classes, better for imbalanced datasets
f1 score	$2 \cdot \frac{precision\dot{r}ecall}{precision+recall}$	good measure if we seek for a trade-of between precision and recall, we migh compare more classifiers using it

Appendix B

CAPEv2 sandbox details

List of file types

- PE files
- DLL files
- PDF documents
- Microsoft Office documents
- URLs and HTML files (even internet explorer behaviour after opening some URL)
- PHP scripts
- CPL files
- Visual Basic scripts
- ZIP files
- Java JAR or applets
- Python files
- PowerShell scripts
- Microsoft windows installer
- Generic binary data such as shellcodes

Table B.1: Parts of *report.json*

Entry	Note
statistics	time statistics for particular part of malware analysis
info	sandbox details (machine, category, used module, timeout etc.)
debug	sandbox debug log
target	info about examined sample
CAPE	extracted payload info
behaviour	processes, mutexes, commands and other behavioural attributes
deduplicated shots	screenshot summary
network	network traffic report (domains, tcp, udp etc.)
static analysis	results per file
strings	extracted strings
suricata	output of suricata network detection tool
malfamily tag	malware family detection result
malscore	malicious score
signatures	list of signatures detected by sandbox

Table B.2: Behaviour parts of *report.json*

Entry	**Meaning**
processes	list of processes related to malware execution with details (API names, arguments)
process tree	structure of process execution
summary	list of occured files, registry keys, mutexes, executed commands, API calls
enhanced	comprehensive log of events during malware execution including parameters, timestamps etc.

Appendix C

Network architecture for distributed sandbox

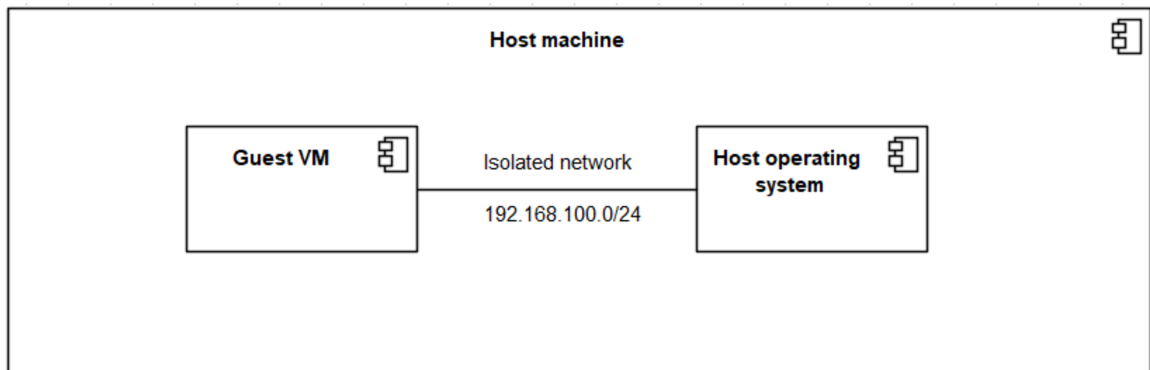


Figure C.1: *None* network setup

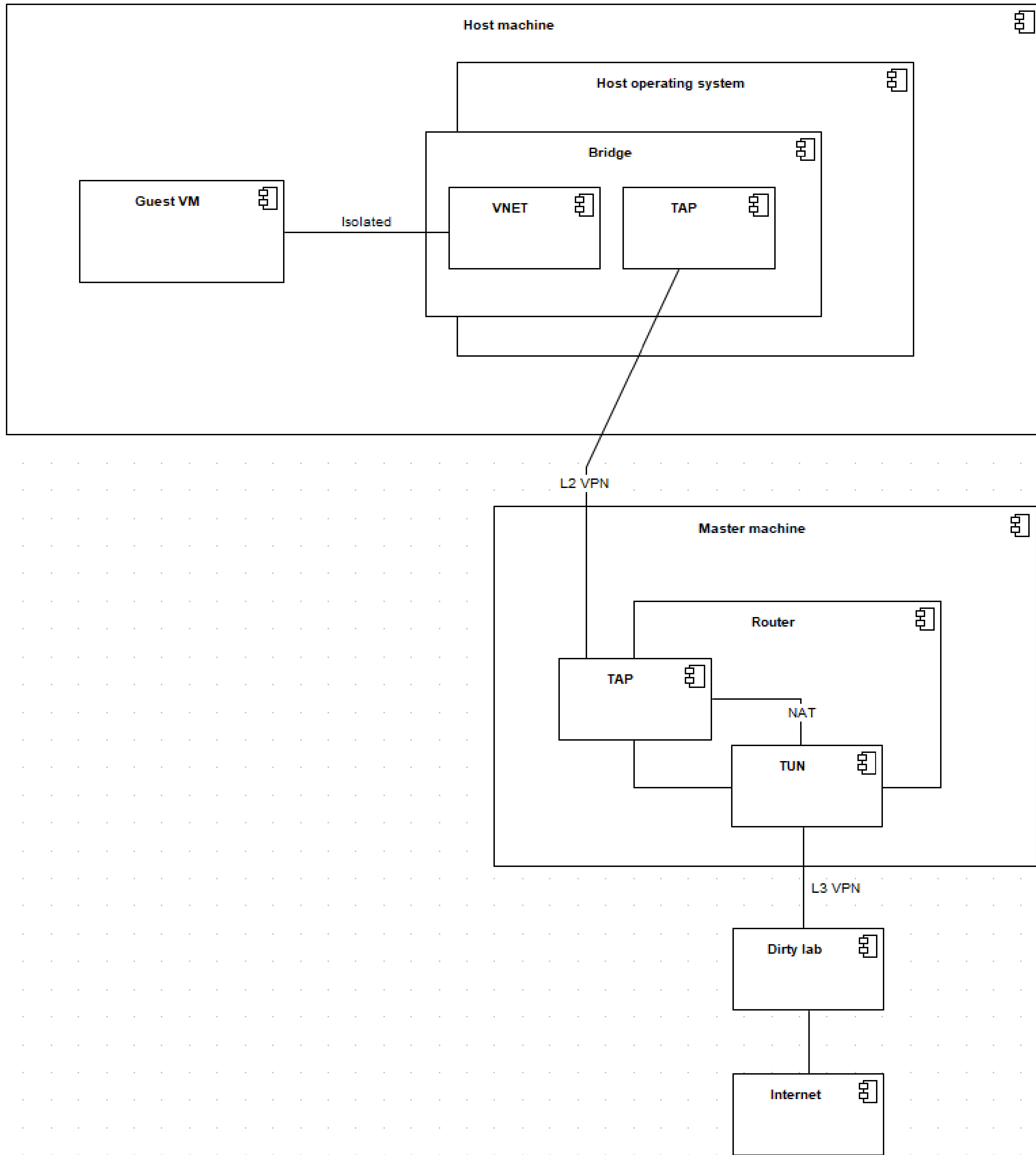


Figure C.2: *Internet* network setup

Appendix D

Signatures description

The split in the table D.1 denotes two groups of signatures described at the end of chapter 4. Signature entry in the original report can be seen in D.1. Implementation of signatures could be seen on https://github.com/kevoreilly/community and an example is in D.2.

```
{
  "name": "dead_connect",
  "description": "Attempts to connect to a dead IP:Port (1
     unique times)",
  "severity": 1,
  "weight": 0,
  "confidence": 100,
  "references": [],
  "data": [
    {
      "IP": "23.238.43.43:80"
    }
  ],
  "new_data": [],
  "alert": false,
  "families": []
},
```

Listing D.1: Example of signature part entry in *report.json*

```
class antidebug_setunhandledexceptionfilter(Signature):
name = "antidebug_setunhandledexceptionfilter"
description = "SetUnhandledExceptionFilter detected (possible
   anti-debug)"
severity = 1
categories = ["anti-debug"]
authors = ["redsand"]
minimum = "1.3"
evented = True

def __init__(self, *args, **kwargs):
    Signature.__init__(self, *args, **kwargs)

filter_apinames = set(["SetUnhandledExceptionFilter"])

def on_call(self, call, process):
    if call["api"] == "SetUnhandledExceptionFilter":
        return True
```

Listing D.2: Example of signature implementation

Table D.1: Used signatures and their details

Signature	Description	Cause	P [%][a]
antidebug setunhandledexceptionfilter	filters api call *SetUnhandledExceptionFilter*, which enables an application to supersede the top-level exception handler of each thread of a process (source https://docs.microsoft.com/)	API CALLS	45
copiesself	detects that currently analysed file copies itself	DROPPED FILES	18
deletesself	detects that currently analysed file deletes/move itself or directory where placed, examining parameters of the call	API CALLS, basic file attributes	27
enumerates running processes	detects more than five process detail listings, saves *pids* in the data part	API CALLS	16
stealthtimeout	detects a sequence of API calls which seems like expiration check and premature exit	API CALLS	21
useswindowsutilities	detects usage of usual windows utilities (*attrib, copy, dir, echo, erase...*)	COMMANDS	18
removeszoneidads	detects attempts to remove an evidence of file downloaded from the internet by examining parameters of API calls	API CALLS	28
antisandboxsleep	detects attempts to delay the analysis task, saves *pids* and time to data part	TIME, API CALLS	39
dropper	detects dropping of a binary and its execution	PROCESSES, DROPPED FILES	15
invalid authenticode signature	detects that the authenticode signature is invalid	STATIC, DROPPED FILES	36
packerentropy	detects encrypted or compressed data using entropy calculation	STATIC	22
stealthnetwork	detects network activity which is not expressed in API calls	NETWORK	66

[a]positive examples ratio in our dataset (80000 samples)

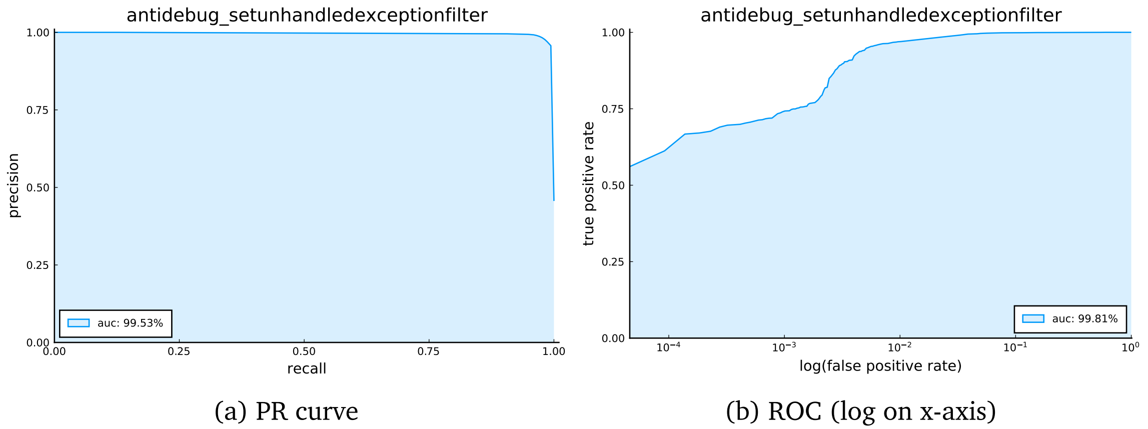
Appendix E

Model metrics

Table E.1: Additional classifier results for particular signatures (rounded off to 3 decimal digits)

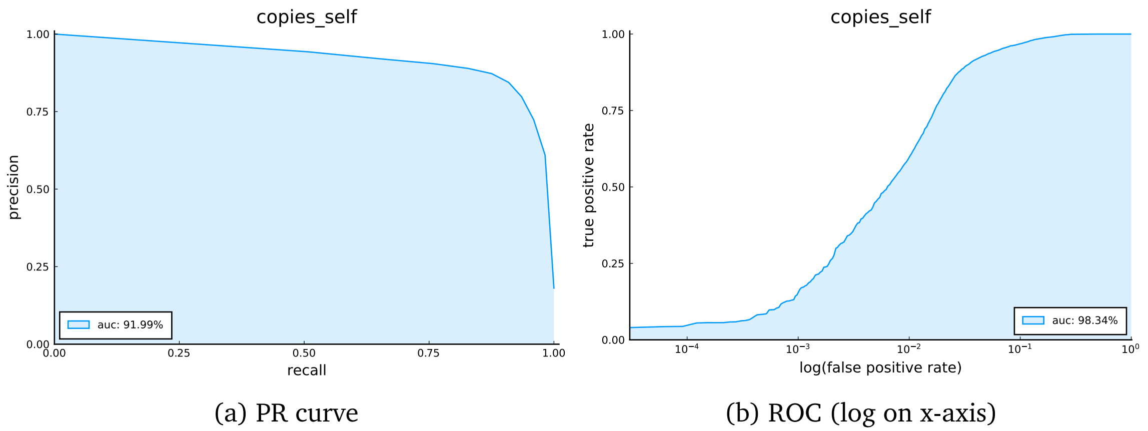
signature	f1	AUROC	AUPRC	test loss[a]
antidebug setunhandledexceptionfilter	0.979	0.998	0.871	0.054
copiesself	0.874	0.983	0.920	0.132
deletesself	0.995	0.999	0.9997	0.008
enumerates running processes	0.957	0.996	0.989	0.041
stealthtimeout	0.802	0.794	0.929	0.263
useswindowsutilities	0.945	0.996	0.987	0.057
removeszoneidads	1.00	1.00	1.00	0.00
antisandboxsleep	0.962	0.993	0.986	0.087
dropper	0.840	0.982	0.902	0.125
invalid authenticode signature	0.433	0.714	0.608	0.569
packerentropy	0.359	0.776	0.518	0.436
stealthnetwork	0.969	0.978	0.987	0.138

[a] logit binary cross entropy



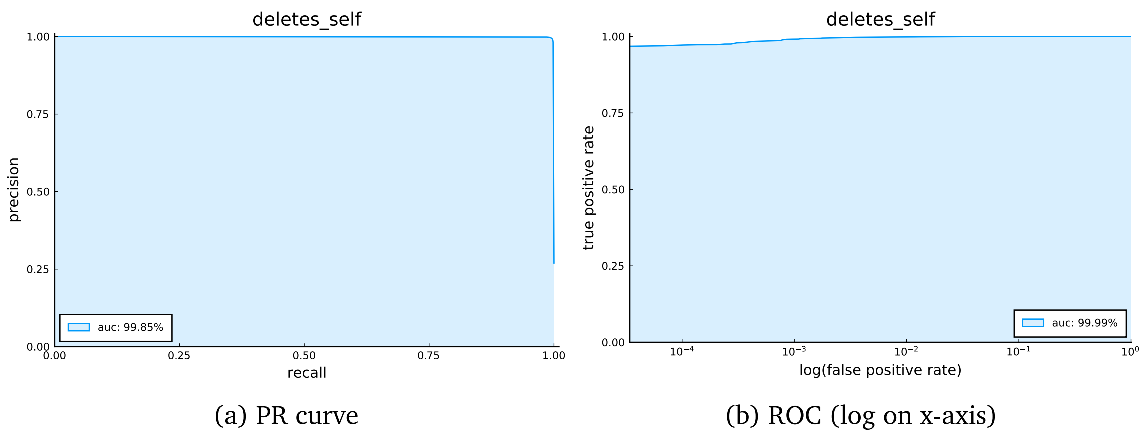
(a) PR curve (b) ROC (log on x-axis)

Figure E.1: antidebug setunhandledexceptionfilter plots



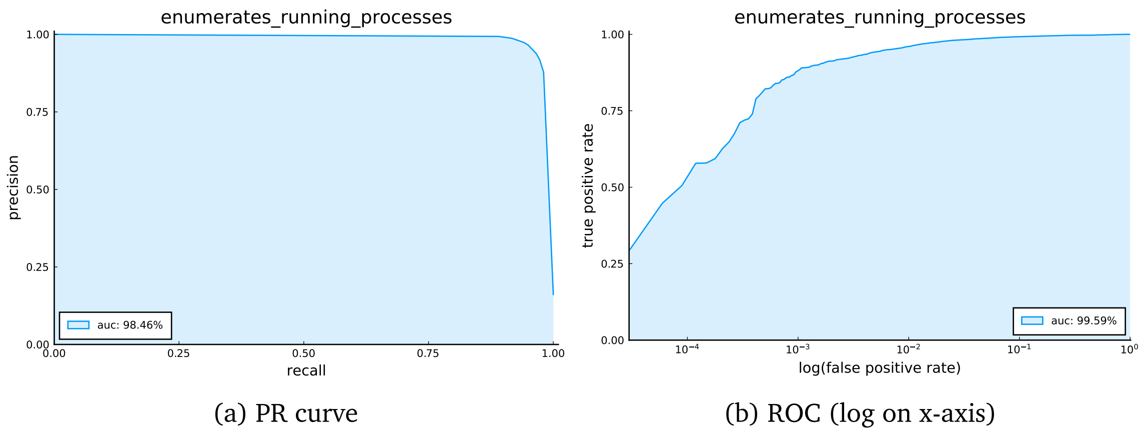
(a) PR curve (b) ROC (log on x-axis)

Figure E.2: copiesself plots



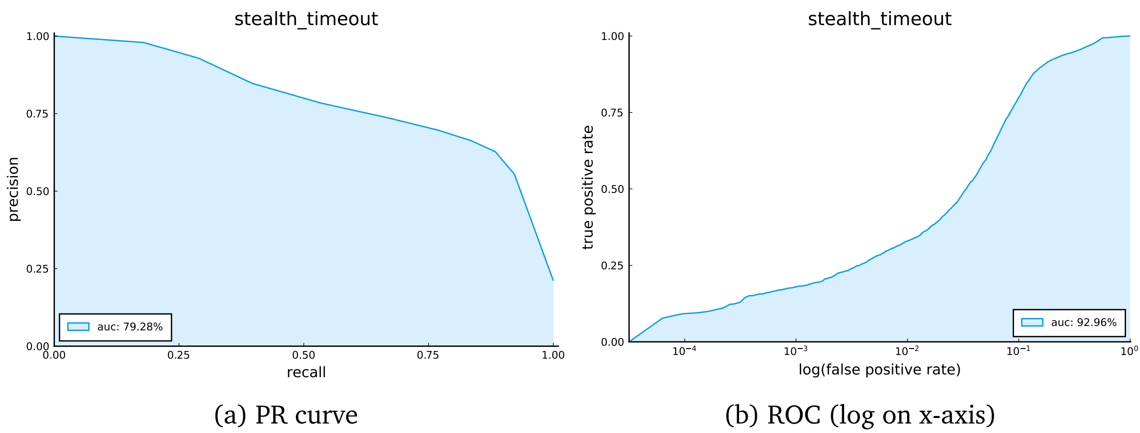
(a) PR curve (b) ROC (log on x-axis)

Figure E.3: deletesself plots



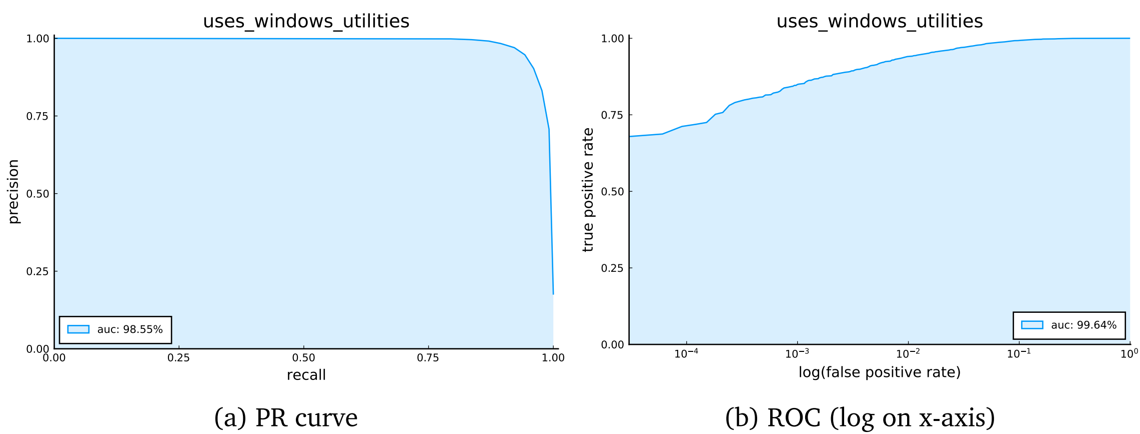
(a) PR curve (b) ROC (log on x-axis)

Figure E.4: enumerates running processes plots



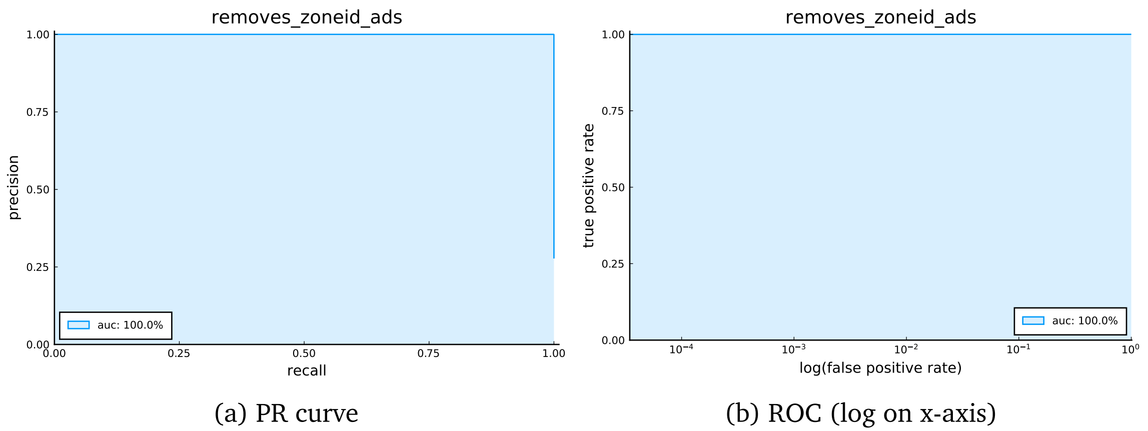
(a) PR curve (b) ROC (log on x-axis)

Figure E.5: stealthtimeout plots



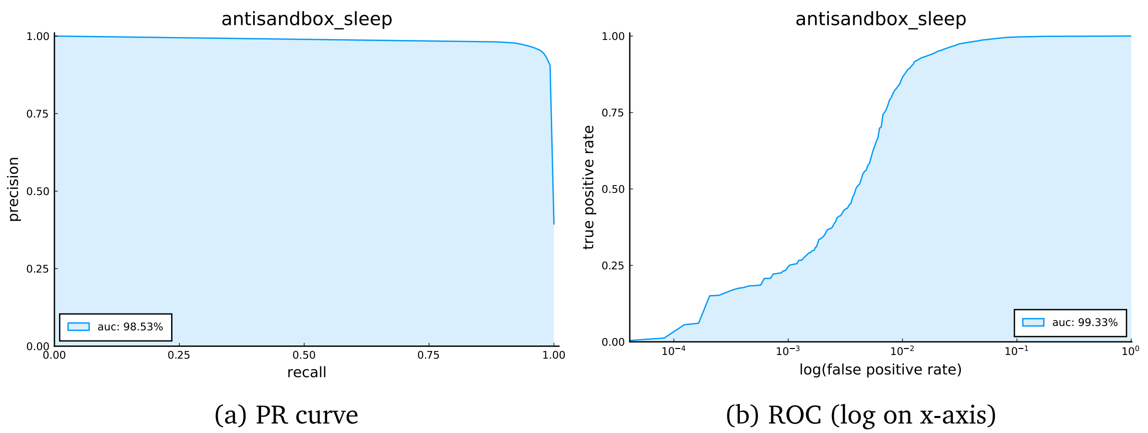
(a) PR curve (b) ROC (log on x-axis)

Figure E.6: uses windows utilities plots



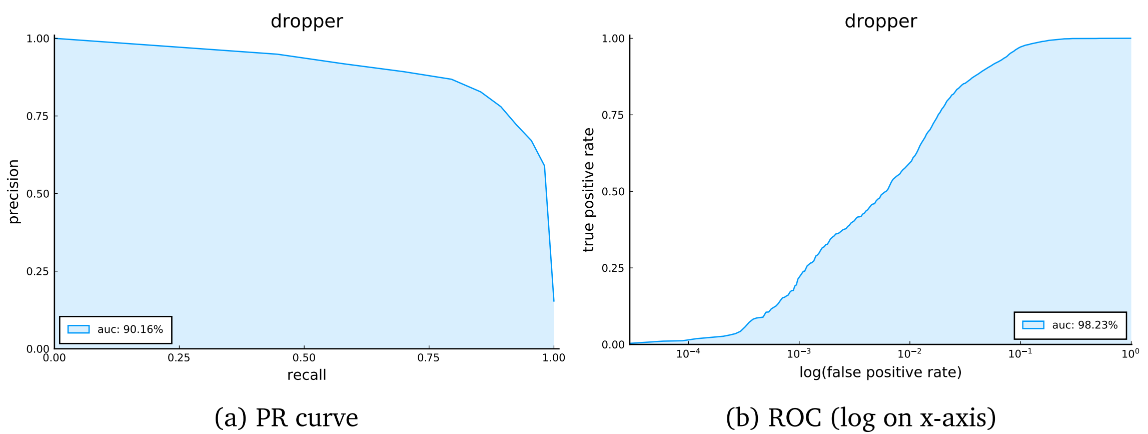
(a) PR curve (b) ROC (log on x-axis)

Figure E.7: removeszoneidads plots



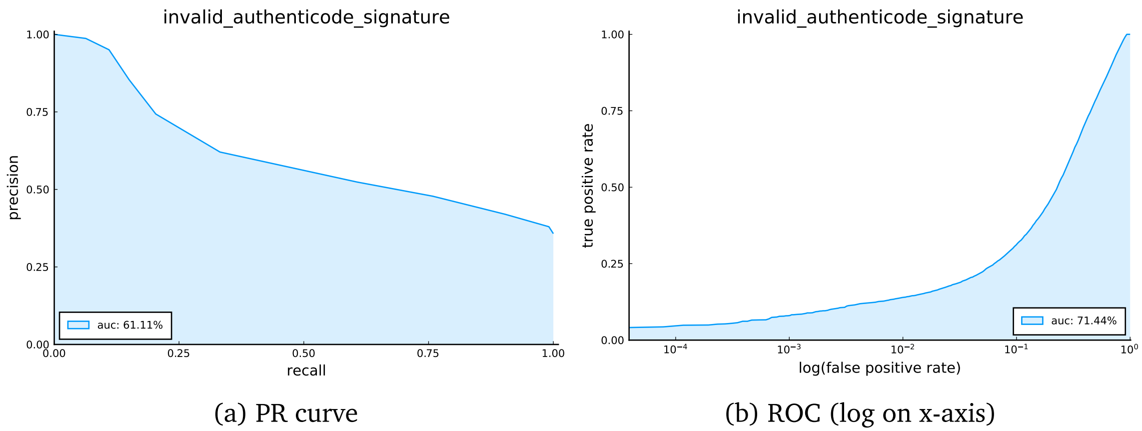
(a) PR curve (b) ROC (log on x-axis)

Figure E.8: antisandboxsleep plots



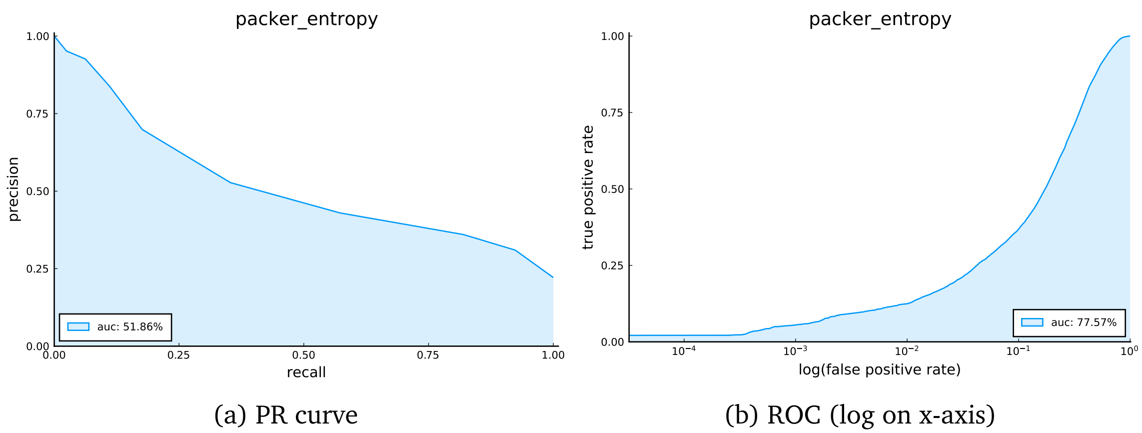
(a) PR curve (b) ROC (log on x-axis)

Figure E.9: dropper plots



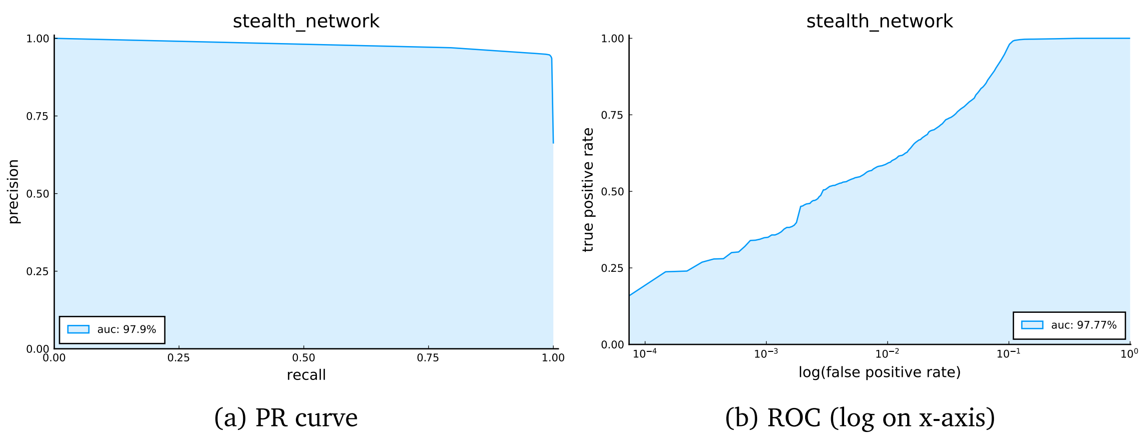
(a) PR curve (b) ROC (log on x-axis)

Figure E.10: invalid authenticode signature plots



(a) PR curve (b) ROC (log on x-axis)

Figure E.11: packer entropy plots



(a) PR curve (b) ROC (log on x-axis)

Figure E.12: stealth network plots

Appendix F

Explaining details

Explanations themselves are in the attachment H. We add some details about the results of explanation in the table F.1, where N denotes the number of explanations, and AS denotes the average size of the explanation. Results are not available where *NaN* is presented — in the first run, we do not have two signatures because their models were not trained at the time we created this explanation, two models were not explained because of their low overall confidence.

Table F.1: Details for each signature's model explanation

signature	N1	AS1	N2	AS2
antidebug setunhandledexceptionfilter	64	3.00	377	3.14
copiesself	4	3.00	100	2.74
deletesself	50	3.22	100	3.08
enumerates running processes	16	4.75	84	3.33
stealthtimeout	NaN	NaN	78	3.17
useswindowsutilities	NaN	NaN	67	1.86
removeszoneidads	56	3.14	100	3.05
antisandboxsleep	21	4.52	100	3.32
dropper	9	3.33	71	4.30
invalid authenticode signature	NaN	NaN	NaN	NaN
packerentropy	NaN	NaN	NaN	NaN
stealthnetwork	44	2.95	100	2.72

Appendix G

Technology

Our technology stack is quite broad, in the lists below, we divide the tools into two groups based on what we did use it for. We do not mention standard Linux tools like *shell scripts*, *ssh* etc. We mention only those which had a significant impact on our work.

Sandboxing and infrastructure

Capev2 with community signatures

There are several relevant sources, here are the most important:

- Public instance — https://capesandbox.com/
- Opensource project — https://github.com/kevoreilly/CAPEv2 (under GNU General Public License v3.0)
- Community extensions — https://github.com/kevoreilly/community
- Documentation — https://capev2.readthedocs.io/en/latest/

Virtualization

Virtualization of sandbox machines, router machine, and other related stuff was ensured by the following tools:

- *Kernel-based Virtual Machine* — https://www.linux-kvm.org
- *VirtualBox* — https://www.virtualbox.org/
- *Windows 7* — operating system running on sandbox virtual machines
- *Ubuntu server* — operating system running on VPN lab edge router
- Other tools and sources — https://github.com/doomedraven

Networking

- Ansible — https://www.ansible.com/
- OpenVPN — https://openvpn.net/
- brctl — https://linux.die.net/man/8/brctl
- rsyslog — https://www.rsyslog.com/
- fail2ban — https://www.fail2ban.org/
- aide — https://aide.github.io/
- ufw — https://help.ubuntu.com/community/UFW

Programming

For programming tasks in the infrastructure part we used *Python 3* (see in H).

Others

We are really pleased that we could use *pafish* (https://github.com/a0rtega/pafish) as a testing malware sample, we used it many times.

Data and machine learning

Julia

For programming tasks in this part we used *Julia* language — https://julialang.org/. Julia has many advantages regarding maily performance compared to Python which could be considered as alternative. But we do not aspire to advocate this language, programming environment was mainly determined by the *HMill* framework which is implemented in this language by [68]. List of the most important used libraries and their versions:

- *JsonGrinder.jl* — [82] (v2.1.4)
- *Mill.jl* — [81] (v2.4.1)
- *Flux.jl* — [49, 50] (v0.11.6)
- *EvalMetrics.jl* — https://github.com/VaclavMacha/EvalMetrics.jL (v0.2.1)

Computing grid system

For resource-demanding computation we used CESNET metacenter (mentioned in acknowledgements) — https://www.metacentrum.cz/en/Sluzby/Grid/index.html.

Appendix H

Attachments

Code

INFRASTRUCTURE	
init_cape_machine.sh	set up a new machine with *Ubuntu 20.04* with the sandbox and all requirements like *KVM*, network security
ansible_init_cape.yml	set up multiple machines at one time using ansible
fetch_samples.py	fetch malware samples from defined sources (abuse.ch)
fetch_metadata.py	fetch metadata for already retrieved samples from defined sources (VirusTotal)
filter_samples.py	filter retrieved samples by filetype
distribute_samples.py	distribute samples on multiple instances of *CAPEv2* sandbox
collect_results.py	collect results of analyses
aggregate_results.py	aggregate results of analyses on master machine
dbutils.py	create new samples in database and other tools for database maintenance
dbs/filedb.json	example of json database of files
dbs/machinedb.json	example of json database of physical machines in cluster

DATA	
data_prune.jl	prune json files using lazy json loading
binary.jl	binary classifier
explain_binary.jl	explainer of binary classification model
grid_submit.sh	script used to schedule job on grid cluster
merge_explanations.py	merge multiple explanations into one aggregated json file
plots.jl	generate plots for classifier (*ROC, PRC*)

Results

Explanations:

- original explanations (2 runs) as json file which includes list of jsons representing explanation for each signature
- *merged* explanations where jsons are aggregated into one and for each entry we have frequency in the original explanation list, in second run we have even *mergedtop* where we have only frequencies of ten most seen entries
- *frequencies* of keys in explanations
- *overall* explanation report to see the intersection of entries across signatures

Data

In attachment we can also find an example of many time referenced *report.json* and an example of *HMill* schemata, extractor and model in text form.

Acronyms

HMill	Hierarchical Multiple Instance Learning
RAM	Random Access Memory
JSON	Javascript Object Notation
API	Application Programming Interface
FNR	False Negative Rate
FPR	False Positive Rate
ROC	Receiver Operating Characteristic
PRC	Precision Recall Curve
AUC	Area Under the Curve
p.d.f.	Probability Density Function
i.i.d.	Independent and Identically Distributed
MLE	Maximum Likelihood Estimation
IoT	Internet of Things
DDOS	Distributed Denial of Service
C2	Command and Control
HTML	Hypertext Markup Language
IAT	Import Address Table
CLI	Command Line Interface
GUI	Graphical User Interface
NAT	Network Address Translation
VPN	Virtual Private Network
PDF	Portable Document Format
PE	Portable Executable
DLL	Dynamic-link Library
URL	Uniform Resource Locator

CPL	Control Panel
JAR	Java Archive
OS	Operating System
KVM	Kernel-based Virtual Machine
HTTP	Hypertext Transfer Protocol
REST	Representational State Transfer
KNN	K-Nearest Neighbors
SVM	Support Vector Machine
GDPR	General Data Protection Regulation
AI	Artificial Intelligence
SSH	Secure Shell
SSD	Solid State Drive
IP	Internet Protocol
MAC	Media Access Control
NAS	Network Attached Storage
IRC	Internet Relay Chat
SMTP	Simple Mail Transfer Protocol
BLAS	Basic Linear Algebra Subprograms
CPU	Central Processing Unit

Bibliography

[1] I. Abdessadki and S. Lazaar. A New Classification Based Model for Malicious PE Files Detection. *International Journal of Computer Network and Information Security*, 11:1–9, 2019.

[2] F. Afghah, A. Razi, R. Soroushmehr, H. Ghanbari, and K. Najarian. Game Theoretic Approach for Systematic Feature Selection; Application in False Alarm Detection in Intensive Care Units. *Entropy*, 20(3), 2018.

[3] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste. Malware Dynamic Analysis Evasion Techniques: A Survey, 2018.

[4] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto. Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification, 2016.

[5] D. Alvarez-Melis and T. S. Jaakkola. Towards Robust Interpretability with Self-Explaining Neural Networks, 2018.

[6] J. Amores. Multiple instance classification: Review, taxonomy and comparative study, 2013.

[7] S. Andrews, I. Tsochantaridis, and T. Hofmann. Support Vector Machines for Multiple-Instance Learning. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems*, volume 15. MIT Press, 2003.

[8] G. Apruzzese, M. Colajanni, L. Ferretti, A. Guido, and M. Marchetti. On the effectiveness of machine and deep learning for cyber security. pages 371–390, 2018.

[9] L. Arras, F. Horn, G. Montavon, K.-R. Müller, and W. Samek. “What is relevant in a text document?”: An interpretable machine learning approach. *PLOS ONE*, 12(8):e0181142, 2017.

[10] A. B. Arrieta, N. Díaz-Rodríguez, J. D. Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-López, D. Molina, R. Benjamins, R. Chatila, and F. Herrera. Explainable Artificial Intelligence (XAI): Concepts, Taxonomies, Opportunities and Challenges toward Responsible AI, 2019.

[11] Y. Bachrach, E. Markakis, E. Resnick, A. Procaccia, J. Rosenschein, and A. Saberi. Approximating power indices: Theoretical and empirical analysis. *Autonomous Agents and Multi-Agent Systems*, 20:105–122, 2010.

[12] E. Boldyreva. Cambridge Analytica: Ethics And Online Manipulation With Decision-Making Process. pages 91–102, 2018.

[13] K. M. Borgwardt, UnderlineCS, S. Schönauer, S. V. N. Vishwanathan, A. Smola, and H. P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 21 Suppl 1:i47–56, 2005.

[14] A. Boukhtouta, S. Mokhov, N.-E. Lakhdari, M. Debbabi, and J. Paquet. Network Malware Classification Comparison Using DPI and Flow Packet Headers. *Journal of Computer Virology and Hacking Techniques*, 11:1–32, 2015.

[15] A. Brukhovetskyy and K. O'Reilly. CAPE Sandbox v2.1 Book. `https://capev2.readthedocs.io/en/latest/index.html`, 2020. (Accessed on 04/29/2021).

[16] R. C. Bunescu and R. J. Mooney. Multiple Instance Learning for Sparse Positive Bags. In *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 105–112, New York, NY, USA, 2007. Association for Computing Machinery.

[17] D. Carlin, A. Cowan, P. O'Kane, and S. Sezer. The Effects of Traditional Anti-Virus Labels on Malware Detection Using Dynamic Runtime Opcodes. *IEEE Access*, 5:17742–17752, 2017.

[18] R. Caruana, H. Kangarloo, J. Dionisio, U. Sinha, and D. Johnson. Case-based explanation of non-case-based learning methods. *Proceedings / AMIA ... Annual Symposium. AMIA Symposium*, pages 212–215, 1999.

[19] A. Chailytko and S. Skuratovich. Defeating Sandbox Evasion: How To Increase The Successful Emulation Rate In Your Virtual Environment. `https://blog.checkpoint.com/wp-content/uploads/2016/10/DefeatingSandBoxEvasion-VB2016_CheckPoint.pdf`, 2019. (Accessed on 05/01/2021).

[20] V. Chandola, A. Banerjee, and V. Kumar. Anomaly Detection: A Survey. *ACM Comput. Surv.*, 41(3), 2009.

[21] C. Cimpanu. First death reported following a ransomware attack on a German hospital. `https://www.zdnet.com/article/first-death-reported-following-a-ransomware-attack-on-a-german-hospital/`, 2020. (Accessed on 04/27/2021).

[22] R. Cohen and D. Walkowski. Banking Trojans: A Reference Guide to the Malware Family Tree. `https://www.f5.com/labs/articles/education/banking-trojans-a-reference-guide-to-the-malware-family-tree`, 2019. (Accessed on 04/28/2021).

[23] E. Cole. *Network Security Bible*. Wiley Publishing, 2nd edition, 2009.

[24] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, 39(1):1–38, 1977.

[25] T. G. Dietterich, R. H. Lathrop, and T. Lozano-Pérez. Solving the Multiple Instance Problem with Axis-Parallel Rectangles. *Artif. Intell.*, 89(1–2):31–71, 1997.

[26] L. Dong. *A Comparison of Multi-instance Learning Algorithms*. PhD thesis, Hamilton, New Zealand, 2006.

[27] H. Edwards and A. Storkey. Towards a Neural Statistician, 2017.

[28] D. Erceg-Hurn and V. Mirosevich. Modern Robust Statistical Methods An Easy Way to Maximize the Accuracy and Power of Your Research. *The American psychologist*, 63:591–601, 2008.

[29] T. Fawcett. Introduction to ROC analysis. *Pattern Recognition Letters*, 27:861–874, 2006.

[30] P. Flach and M. Kull. Precision-Recall-Gain Curves: PR Analysis Done Right. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[31] V. Franc, B. Flach, and J. Drchal. Lecture notes in Statistical Machine Learning, 2020.

[32] E. Frank and X. Xu. Applying propositional learning algorithms to multi-instance data. Technical report, 2003.

[33] J. Fruhlinger. Top cybersecurity facts, figures and statistics. `https://www.csoonline.com/article/3153707/top-cybersecurity-facts-figures-and-statistics.html`, 2020. (Accessed on 04/27/2021).

[34] Z. Fuyong and Z. Tiezhu. Malware Detection and Classification Based on N-Grams Attribute Similarity. pages 793–796, 2017.

[35] H. Galal. Behavior-based features model for malware detection. *Journal of Computer Virology and Hacking Techniques*, 12, 2015.

[36] T. Gärtner, P. Flach, A. Kowalczyk, and A. Smola. Multi-Instance Kernels. pages 179–186, 2002.

[37] C. Gentile and M. Warmuth. Linear Hinge Loss and Average Margin. In *Proc. Adv. Neural Inform. Processing Systems (NIPS)*, pages 225–231, 1998.

[38] M. Ghiasi, A. Sami, and Z. Salehi. Dynamic VSA: a framework for malware detection based on register contents. *Engineering Applications of Artificial Intelligence*, 44:111–122, 2015.

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

[40] I. J. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, Cambridge, MA, USA, 2016.

[41] J. Graham, R. Olson, and R. Howard. *Cyber Security Essentials*. Auerbach Publications, USA, 1st edition, 2010.

[42] R. Guidotti, A. Monreale, F. Turini, D. Pedreschi, and F. Giannotti. A Survey Of Methods For Explaining Black Box Models. *CoRR*, abs/1802.0, 2018.

[43] J. Haugeland. *Artificial Intelligence: The Very Idea*. Cambridge: MIT Press, 1985.

[44] M. Henaff, J. Bruna, and Y. Lecun. Deep Convolutional Networks on Graph-Structured Data. 2015.

[45] L. Hernández-Callejo, S. Egea, and A. Sánchez-Esguevillas. Exploratory Study on Class Imbalance and Solutions for Network Traffic Classification. *Neurocomputing*, 343, 2019.

[46] K. Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Netw.*, 4(2):251–257, Mar. 1991.

[47] Q. Huang, M. Yamada, Y. Tian, D. Singh, D. Yin, and Y. Chang. GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks, 2020.

[48] F. Iglesias Vázquez and T. Zseby. Analysis of network traffic features for anomaly detection. *Machine Learning*, 101, 2014.

[49] M. Innes. Flux: Elegant Machine Learning with Julia. *Journal of Open Source Software*, 2018.

[50] M. Innes, E. Saba, K. Fischer, D. Gandhi, M. C. Rudilosso, N. M. Joy, T. Karmali, A. Pal, and V. Shah. Fashionable Modelling with Flux. *CoRR*, abs/1811.0, 2018.

[51] V. Jaganathan, P. Cherurveettil, and P. Sivashanmugam. Using a Prediction Model to Manage Cyber Security Threats. *The Scientific World Journal*, 2015:1–5, 2015.

[52] J. Janisch, T. Pevný, and V. Lisý. Cost-Efficient Hierarchical Knowledge Extraction with Deep Reinforcement Learning, 2020.

[53] M. K A. *Learning Malware Analysis: Explore the Concepts, Tools, and Techniques to Analyze and Investigate Windows Malware*. Packt Publishing, Limited, Birmingham, 2018.

[54] J. Keeler, D. Rumelhart, and W. Leow. Integrated Segmentation and Recognition of Hand-Printed Numerals. In R. P. Lippmann, J. Moody, and D. Touretzky, editors, *Advances in Neural Information Processing Systems*, volume 3. Morgan-Kaufmann, 1991.

[55] D. Kenny. *Correlation and Causality*, volume -1. 1979.

[56] J. Kiefer and J. Wolfowitz. Stochastic Estimation of the Maximum of a Regression Function. *The Annals of Mathematical Statistics*, 23(3):462–466, 1952.

[57] D. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations*, 2014.

[58] S. Krening, B. Harrison, K. M. Feigh, C. L. Isbell, M. Riedl, and A. Thomaz. Learning From Explanations Using Sentiment and Advice in RL. *IEEE Transactions on Cognitive and Developmental Systems*, 9(1):44–55, 2017.

[59] M. Kruczkowski and E. N. Szynkiewicz. Support Vector Machine for Malware Analysis and Classification. In *Proceedings of the 2014 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT) - Volume 02*, WI-IAT '14, pages 415–420, USA, 2014. IEEE Computer Society.

[60] O. Kubovič. False positives can be more costly than a malware infection. `https://www.welivesecurity.com/2017/05/09/false-positives-can-costly-malware-infection/`, 2017. (Accessed on 04/04/2021).

[61] J. Kwon and H. Lee. BinGraph: Discovering mutant malware using hierarchical semantic signatures. In *2012 7th International Conference on Malicious and Unwanted Software*, pages 104–111, 2012.

[62] W. Landecker, M. D. Thomure, L. Bettencourt, M. Mitchell, G. Kenyon, and S. Brumby. Interpreting individual classifications of hierarchical networks. pages 32–38, 2013.

[63] S. Lapuschkin, A. Binder, G. Montavon, F. Klauschen, K.-R. Müller, and W. Samek. On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation. *PLoS ONE*, 10:e0130140, 2015.

[64] J. Lee, C. Im, and H. Jeong. A study of malware detection and classification by comparing extracted strings. page 75, 2011.

[65] J. Li, X. Chen, E. Hovy, and D. Jurafsky. Visualizing and Understanding Neural Models in {NLP}. In *Proceedings of the 2016 Conference of the North {A}merican Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 681–691, San Diego, California, 2016. Association for Computational Linguistics.

[66] Z. C. Lipton. The Mythos of Model Interpretability. *CoRR*, abs/1606.0, 2016.

[67] G. D. Maayan. The IoT Rundown For 2020: Stats, Risks, and Solutions – Security Today. `https://securitytoday.com/Articles/2020/01/13/The-IoT-Rundown-for-2020.aspx`, 2020. (Accessed on 04/27/2021).

[68] S. Mandlik and T. Pevny. Mapping the Internet — Modelling Entity Interactions in Complex Heterogeneous Networks. Master's thesis, 2020.

[69] O. Maron and T. Lozano-Pérez. A Framework for Multiple-Instance Learning. In M. Jordan, M. Kearns, and S. Solla, editors, *Advances in Neural Information Processing Systems*, volume 10. MIT Press, 1998.

[70] Microsoft. PE Format. `https://docs.microsoft.com/en-us/windows/win32/debug/pe-format`, 2021. (Accessed on 04/29/2021).

[71] G. Montavon, W. Samek, and K.-R. Müller. Methods for interpreting and understanding deep neural networks. *Digital Signal Processing*, 73:1–15, 2018.

[72] K. P. Murphy. *Machine learning : a probabilistic perspective*. MIT Press, Cambridge, Mass. [u.a.], 2013.

[73] E. Nowak, F. Jurie, and B. Triggs. Sampling Strategies for Bag-of-Features Image Classification. In A. Leonardis, H. Bischof, and A. Pinz, editors, *Computer Vision – ECCV 2006*, pages 490–503, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[74] D. Oktavianto and I. Muhardianto. *Cuckoo Malware Analysis*. Packt Publishing, 2013.

[75] Open Web Application Security Project. Intrusion Detection. `https://owasp.org/www-community/controls/Intrusion_Detection`, 2020. (Accessed on 04/04/2021).

[76] K. O'Reilly. CAPEv2. `https://github.com/kevoreilly/CAPEv2`, 2020. Accessed on 06/30/2020.

[77] K. Pearson. On Lines and Planes of Closest Fit to Systems of Points in Space. *Philosophical Magazine*, 2(6):559–572, 1901.

[78] T. Pevný. Explaining Classifiers Trained on Raw Hierarchical Multiple-Instance Data. 2020.

[79] T. Pevny and M. Dedic. Nested Multiple Instance Learning in Modelling of HTTP network traffic, 2020.

[80] T. Pevny and V. Kovarik. Approximation capability of neural networks on spaces of probability measures and tree-structured domains, 2019.

[81] T. Pevný and Š. Mandlík. Mill.jl framework: a flexible library for (hierarchical) multi-instance learning. `https://github.com/CTUAvastLab/Mill.jl`, 2018. (Accessed on 01/01/2021).

[82] T. Pevný and M. Račinský. JsonGrinder.jl. `https://github.com/CTUAvastLab/JsonGrinder.jl`, 2019. (Accessed on 01/01/2021).

[83] T. Pevny and P. Somol. Discriminative Models for Multi-Instance Problems with Tree Structure. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, AISec '16, pages 83–91, New York, NY, USA, 2016. Association for Computing Machinery.

[84] T. Pevný and P. Somol. Using Neural Network Formalism to Solve Multiple-Instance Problems. In F. Cong, A. Leung, and Q. Wei, editors, *Advances in Neural Networks - ISNN 2017*, pages 135–142, Cham, 2017. Springer International Publishing.

[85] PurpleSec LLC. Cyber Security Trends In 2021. `https://purplesec.us/cyber-security-trends-2021/`, 2021. (Accessed on 04/27/2021).

[86] M. Robnik-Šikonja and I. Kononenko. Explaining Classifications For Individual Instances. *IEEE Transactions on Knowledge and Data Engineering*, 20(5):589–600, 2008.

[87] T. Roccia and C. Shah. Evolution of Malware Sandbox Evasion Tactics – A Retrospective Study. `https://www.mcafee.com/blogs/other-blogs/mcafee-labs/evolution-of-malware-sandbox-evasion-tactics-a-retrospective-study/`, 2019. (Accessed on 04/29/2021).

[88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning Representations by Back-Propagating Errors*, pages 696–699. MIT Press, Cambridge, MA, USA, 1988.

[89] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, USA, 3rd edition, 2009.

[90] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini. The Graph Neural Network Model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

[91] K. Sethi, R. Kumar, L. Sethi, P. Bera, and P. K. Patra. A Novel Machine Learning Based Malware Detection and Classification Framework. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, pages 1–4, 2019.

[92] K. Siau and W. Wang. Artificial Intelligence (AI) Ethics: Ethics of AI and Ethical AI. *Journal of Database Management*, 31:74–87, 2020.

[93] M. Sikorski and A. Honig. *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, USA, 1st edition, 2012.

[94] K. Simonyan, A. Vedaldi, and A. Zisserman. Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps, 2014.

[95] J. Singh and J. Singh. A survey on machine learning-based malware detection in executable files. *Journal of Systems Architecture*, 112:101861, 2020.

[96] A. Skelly, J. Dettori, and E. Brodt. Assessing bias: the importance of considering confounding. *Evidence-based spine-care journal*, 3:9–12, 2012.

[97] E. Štrumbelj and I. Kononenko. Explaining prediction models and individual predictions with feature contributions. *Knowledge and Information Systems*, 41:647–665, 2013.

[98] The Independent IT-Security Institute. AV-ATLAS - Malware & PUA. `https://portal.av-atlas.org/malware`, 2021. (Accessed on 04/27/2021).

[99] R. van Zutphen. Cuckoo Sandbox Architecture. `https://hatching.io/blog/cuckoo-sandbox-architecture/`, 2019. (Accessed on 04/30/2021).

[100] J. Wang and J.-d. Zucker. Solving the multiple-instance problem: A lazy learning approach. In *Proc. 17th International Con. on Machine Learning*, pages 1119–1126, 2000.

[101] M. Wojnowicz, G. Chisholm, M. Wolff, and X. Zhao. Wavelet decomposition of software entropy reveals symptoms of malicious code, 2018.

[102] W. Woof and K. Chen. A Framework for End-to-End Learning on Semantic Tree-Structured Data, 2020.

[103] X. Xu. Statistical learning in multiple instance problems. Technical report, 2003.

[104] R. Ying, D. Bourgeois, J. You, M. Zitnik, and J. Leskovec. GNNExplainer: Generating Explanations for Graph Neural Networks, 2019.

[105] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola. Deep Sets, 2018.

[106] M. D. Zeiler and R. Fergus. Visualizing and Understanding Convolutional Networks, 2013.

[107] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola. *Dive into Deep Learning*. 2020.

[108] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals. Understanding deep learning requires rethinking generalization. *CoRR*, abs/1611.0, 2016.

[109] H. Zimmermann. {OSI} Reference Model-the {ISO} model of architecture for open systems interconnection. *IEEE Trans. Communication (USA)*, COM-28(4):425–432, 1980.